

---

ComponentOne

# FlexGrid for WinForms

**ComponentOne, a division of GrapeCity**

201 South Highland Avenue, Third Floor  
Pittsburgh, PA 15206 USA

**Website:** <http://www.componentone.com>

**Sales:** [sales@componentone.com](mailto: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 media on which the software is delivered is free from defects in material and workmanship, assuming normal use, for a period of 90 days from the date of purchase. If a defect occurs during this time, you may return the defective media to ComponentOne, along with a dated proof of purchase, and ComponentOne will replace it at no charge. After 90 days, you can obtain a replacement for the defective media by sending it and a check for \$2 5 (to cover postage and handling) to ComponentOne.

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

**Copying and Distribution**

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

## Table of Contents

FlexGrid for WinForms Overview	6
Help with WinForms Edition	6
Differences Between the .NET and ActiveX Versions of C1FlexGrid	6-8
Differences Between the .NET and Mobile Versions of FlexGrid for WinForms	8-10
Differences Between FlexGrid for WinForms and True DBGrid for WinForms	10
Key Features	11-12
Feature Comparison Matrix	13-17
FlexGrid for WinForms Quick Start	18
Step 1 of 3: Creating the FlexGrid for WinForms Application	18
Step 2 of 3: Binding C1FlexGrid to a Data Source	18-19
Step 3 of 3: Customizing C1FlexGrid Settings	19-21
Design-Time Support	22
C1FlexGrid Editors	22
C1FlexGrid Column Editor	22-24
C1FlexGrid Style Editor	24-25
Caption Style and Column Style	25-29
C1FlexGrid Smart Tag	29-30
C1FlexGrid Tasks Menu	30-32
Column Tasks Menu	32-34
Using the C1FlexGrid Control	35-37
Rows and Columns	37-38
Column Sizing	38-39
Star Sizing	39-41
Column Footers	41
Cell Selection	41-42
Cell Ranges	42-43
Cell Images	43
Formatting Cells	43
Cell Content	44-45
Cell Appearance	45-47
Conditional Formatting	47-48
Owner-Drawn Cells	48-50
Editing Cells	50-51
Lists and Combos	51-52

Checkboxes	52-53
Value-Mapped Lists	53-56
Cell Buttons	56-57
Masks	57-59
Validation	59-60
Custom Editors	60-61
Creating Custom Editors	61-62
Edit Mode	62-63
Grouping	63-65
Grouping through Code	65-66
Grouping through GroupPanel	66-68
Merging Cells	68-69
Merged Table Headers	69-71
Merged Data Views	71-72
Spilling Text	72-73
Custom Merging	73
Outlining and Summarizing Data	73
Creating Subtotals	73-76
Creating Custom Subtotal	76-78
Creating Custom Trees	78-81
Creating Outlines and Trees with the C1FlexGrid Control	81-82
Loading the Data	82-83
Creating Node Rows	83-86
Outline Tree	86-88
Adding Subtotals	88-91
Using the Subtotal Method	91-92
Outline Maintenance	92-93
Using the Node class	93-94
Saving, Loading, and Printing	94
Saving and Loading Grids to Text Files	94
Saving and Loading Microsoft Excel Files	94
Loading Grids from Databases	95-96
Printing Grids	96-97
C1FlexGrid Filtering	97-98
AllowFiltering Property	98-100
Managing Filters Programmatically	100-101

Applying Filters Programmatically	101-102
Customizing Filter behavior	102-104
Customizing the Filtering UI	104-105
C1FlexGrid Property Groups	105-106
Data Binding	107
Binding to a Data Source	107-109
Storing and Retrieving Data	109-110
FlexGrid for WinForms Samples	111-119
FlexGrid for WinForms Tutorials	120
Edit Tutorial	120-121
Step 1 of 6: Create the C1FlexGrid Control for the Edit Tutorial	121-124
Step 2 of 6: Set Column Types and Formats	124-126
Step 3 of 6: Incorporate Drop-Down Lists	126-129
Step 4 of 6: Add Data Validation	129-131
Step 5 of 6: Add Clipboard Support	131-132
Step 6 of 6: Include Custom Editors	132-135
Outline Tutorial	135
Step 1 of 5: Create the Controls	135-138
Step 2 of 5: Read the Data and Build the Outline	138-143
Step 3 of 5: Add Custom Mouse and Keyboard Handling	143-144
Step 4 of 5: Allow/Prevent Editing	144
Step 5 of 5: Implement ToolTips	145-146
Data Analysis Tutorial	146-147
Step 1 of 4: Create the C1FlexGrid Control for the Data Analysis Tutorial	147
Step 2 of 4: Initialize and Populate the Grid	147-153
Step 3 of 4: Allow Automatic Sorting	153-154
Step 4 of 4: Include Subtotals and Outline Tree	154-156
FlexGrid for WinForms Task-Based Help	157
Accessing the C1FlexGrid Editors	157
Accessing the C1FlexGrid Column Editor	157
Accessing the C1FlexGrid Style Editor	157-158
Adding Pictures and Text to a Cell	158-159
Adding Row Numbers in a Fixed Column	159-161
Adding Three-Dimensional Text to a Header Row	161-162
Adding Three-Dimensional Text to a Header Row Using Built-In Styles	162-164
Changing the Column Order in the Grid	164-165

Filtering by Value	165-166
Filtering by Condition	166-167
Changing the Filter Language	167-168
Clearing a Tree View	168-170
Clearing C1FlexGrid	170
Clearing Content	170-171
Clearing Styles	171-172
Clearing UserData	172
Clearing Content, Styles, and UserData	172-173
Converting Column Letters to Uppercase	173-174
Customizing Appearance Using Visual Styles	174-177
Entering Only Numbers in a Cell	177-178
Formatting Cells	178
Formatting a Cell as Read-Only	178-179
Formatting a Cell with Decimal Content	179-181
Formatting Cells Based on the Contents	181-183
Formatting the Border Style	183
Formatting the Border Style of the Control	183-186
Formatting the Border Style of the Grid	186-191
Freezing Rows and Columns	191-193
Getting the Width of a Partially Visible Column	193-194
Loading and Saving Open XML Files	194-196
Populating an Unbound Grid with Data	196
Populating a Column with Data	196-197
Populating a Range of Cells with Data	197
Populating a Row with Data	197-198
Populating a Single Cell with Data	198
Transposing Data in Grid	198-199
Restricting Grid Editing	199
Disable Editing for the Entire Grid	199-200
Disable Editing for a Specific Column	200
Disable Editing for a Specific Row	200-201
Restricting Sorting for a Specific Column	201
Searching for Entries in a Column	201-202
Searching for Entries in a Grid	202-204

Setting a Cell's Value to Zero When Users Press the Delete Key	204
Setting Rows As Headers	204-207
Setting the Font of a Single Cell	207-209
Setting the Text Delimiting Character in C1FlexGrid	209-210
Sorting Multiple Columns	210-211
Styling and Appearance	211
Setting the Background Color of Columns and Rows	211-214
Undoing a Sort	214-215
Using Password Entries in C1FlexGrid	215-217
Hiding Characters Already Entered	217-218
Word Wrapping in a Header or Fixed Row	218-219
FlexGrid for WinForms Top Tips	220-228
C1FlexGridClassic Control	229

## FlexGrid for WinForms Overview

**FlexGrid for WinForms** incorporates the latest in data-binding technology and integrates seamlessly with the Microsoft.NET Framework. As a result, you get an easy-to-use, flexible grid control for creating user-friendly interfaces that display, edit, format, organize, summarize, and print tabular data.

The **FlexGrid for WinForms** package consists of two controls:

- **C1FlexGrid Control**

The **C1FlexGrid** control is a powerful, full-featured grid. It will read and write grids from and to compressed binary files or text files (compatible with Microsoft® Access and Excel®). C1FlexGrid provides all the basics plus advanced features such as outline trees, sorting, cell merging, masked editing, translated combo and image lists, and automatic data aggregation.

C1FlexGrid can be used in bound mode, where it displays data from any .NET data source, including ADO.NET and **DataObjects for WinForms**, or in unbound mode, where the grid itself manages the data.

- **C1FlexGridClassic Control**

The **C1FlexGridClassic** control is a control that derives from C1FlexGrid and provides an object model that is virtually 100% identical to the VSFlexGrid ActiveX control. C1FlexGridClassic was developed to allow easy migration of existing **VSFlexGrid** projects.

The source code for C1FlexGridClassic is provided as a sample. You can use it as a reference that shows how to use the C1FlexGrid control as a base class in the development of custom grid controls.

### Getting Started

To get started, review the following topics:

- [Key Features](#)
- [FlexGrid for WinForms Quick Start](#)
- [FlexGrid for WinForms Samples](#)
- [FlexGrid for WinForms Top Tips](#)

## Help with WinForms Edition

### Getting Started

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

## Differences Between the .NET and ActiveX Versions of C1FlexGrid

In the ActiveX product, we supplied several versions of the FlexGrid control (ADO, DAO, unbound, Unicode, and so on). In the .NET product, there are two versions: **C1FlexGrid** and C1FlexGridClassic.

C1FlexGrid is not a simple port of the ActiveX control. It is a brand new grid control, written from the ground up in C#, with the same design principles but with a new object model that is more modern, clean, and powerful than the one in the ActiveX control. The C1FlexGrid control can be bound to ADO.NET data sources or used in unbound mode.

To keep the highest level of source-code compatibility with existing applications, and to make the learning curve as smooth as possible for **VSFlexGrid** users, we also offer the C1FlexGridClassic control.

C1FlexGridClassic is a control that uses C1FlexGrid as a base class and exposes an object model that is virtually identical to the one in **VSFlexGrid**. We supply the source code to C1FlexGridClassic so you can see exactly how to use the new object model. You can also use it as an example and create your own grid using the C1FlexGrid as a base class.



If you are writing new applications, you should use the C1FlexGrid control. If you are porting existing applications that use the **VSFlexGrid** ActiveX control and want to change as little code as possible, then use the C1FlexGridClassic control.

The following table lists the differences between the .NET and ActiveX versions of C1FlexGrid:

	VSFlexGrid (ActiveX)	C1FlexGrid (.NET)
<b>Rows, Cols Collections</b>	<p>The ActiveX control has <b>Rows</b> and <b>Cols</b> properties that are used to get or set the number of rows and columns on the grid. In the C1FlexGrid control, these properties return row and column collections. The collections have read/write properties that return the number of elements and fixed elements in each collection. This is probably the most visible change between the controls. Using the ActiveX control, you would write:</p> <pre>Dim r%, c% c = 1 For r = _flex.FixedRows To _flex.Rows - 1     Debug.Print _flex.TextMatrix(r,c) Next</pre>	<p>Using the C1FlexGrid control, this becomes:</p> <pre>Dim r%, c% c = 1 For r = _flex.Rows.Fixed To _flex.Rows.Count - 1     Debug.Print _flex(r,c) Next</pre>
	Uses the <b>TextMatrix</b> property.	Uses indexers.
<b>Styles</b>	<p>In the ActiveX control, you can customize the appearance of individual cells or cell ranges using the <b>Cell</b> property. For example, to give the second row a red background, you would write:</p> <pre>_flex.Cell(flexcpBackColor, 2, 0, 2, _flex.Cols-1) = vbRed</pre>	<p>The C1FlexGrid control uses a <a href="#">CellStyle</a> object to customize cell appearance. To make the second row red, you would write:</p> <pre>Dim redStyle As CellStyle = _flex.Styles.Add("Red") redStyle.BackColor = Color.Red _flex.Rows(2).Style = redStyle</pre> <p>But this requires three lines of code instead of one! What's the advantage? The main advantage of the new approach is that the new style is an object that can be changed or assigned to new ranges. For example, if you decide to give the red cells a white forecolor and a bold font, you can write:</p> <pre>_flex.Styles("Red").ForeColor = Color.White _flex.Styles("Red").Font = new Font("Arial", 9, FontStyle.Bold)</pre> <p>This will change the appearance of all cells that use the "Red" style. The previous approach would require either (1) clearing all styles and setting everything up again from scratch or (2) scanning all cells in the grid to detect which cells are red, then changing those. CellStyle objects are used consistently throughout the control, so instead of <b>BackColorFixed</b> and</p>


	VSFlexGrid (ActiveX)	C1FlexGrid (.NET)
<b>CellRange</b>	<p>The <b>Cell</b> property is one of the most powerful elements of the <b>VSFlexGrid</b> object model. It allows you to get or set any property of any cell or cell range with a single command. However, handling colors, text, values, and so on. Using a single property means using <i>Variants</i>, and this prevents the compiler from catching many subtle problems in case you make mistakes.</p>	<p><b>ForeColorSel</b> you can now write <code>Styles.Fixed.BackColor</code> and <code>Styles.Highlight.ForeColor</code>.</p> <p>The <b>C1FlexGrid</b> replaces the <b>Cell</b> property with a <b>CellRange</b> object that exposes type-safe properties and methods used to access the properties of a cell range. For example, instead of writing:</p> <pre>_flex.Cell(flexcpPicture, 5, 5, 10, 10) = theImage</pre> <p>You would write:</p> <pre>Dim rg As CellRange rg = _flex.GetCellRange(5, 5, 10, 10)  rg.Image = theImage</pre> <p>The new approach has two significant advantages: · It is type-safe, so if the variable <i>theImage</i> contained a string instead of an image, you would get a compiler error instead of a runtime error. · You get command-completion when writing the code because the types for each property are known.</p>
<b>Typed columns</b>	<p>In the ActiveX version, the <b>ColDataType</b> allowed you to set the type of data that each column contained. This information was used mainly for sorting columns that contained dates or numbers.</p>	<p>The .NET version has a <code>Cols[i].DataType</code> property that determines the type of data the column holds. By default, the <code>DataType</code> for all columns is "object", which means you can store anything in any column. You can set the data type to specific types, however, and the grid will try to coerce any data stored in the grid to the proper type. For example:</p> <pre>_flex.Cols[2].DataType = typeof(int); // Value will be set to 12. _flex[1, 2] = "12"; // Bad value. Fire the GridError event and ignore. _flex[2, 2] = "hello";</pre> <p>This code would assign the integer 12 to cell (1,2). Cell (2,2) would retain its original value, because the string "hello" cannot be converted to an integer. If you want to store values of mixed types in a column, you have two options:</p> <ol style="list-style-type: none"> <li>1. Set the column's <code>DataType</code> property to "object".</li> <li>2. Use the <code>SetData</code> method with the <i>coerce</i> parameter set to <b>False</b> to store a value or object without checking the data type.</li> </ol>

## Differences Between the .NET and Mobile Versions of FlexGrid for WinForms

The mobile version of **FlexGrid for WinForms** allows you to quickly develop Microsoft.NET Compact Framework-based applications for your mobile devices, such as personal digital assistants (PDAs), mobile phones and more. It provides the **C1FlexGrid** control, a full-featured grid to display, edit, format, organize, summarize, and print tabular data, but with a compacted version of the FlexGrid for WinForms object model and feature set.

The following table lists the differences between the .NET and .NET Compact Framework versions of **C1FlexGrid**:

	C1FlexGrid (.NET)	C1FlexGrid (Mobile)
<b>C1FlexGrid Editors</b>	C1FlexGrid Column Editor	N/A
	C1FlexGrid Style Editor	N/A
	Caption Style	N/A
	Column Style	N/A
<b>C1FlexGrid Smart Tag</b>	C1FlexGrid Tasks Menu	<b>About C1FlexGrid</b> is the only option available on this menu.
	Column Tasks Menu	N/A
<b>Column Sizing</b>	At design time, columns can be resized using the <b>Width</b> property which can be set directly in the grid by clicking and dragging the horizontal double arrow that appears at the right edge of a column's header, through the <b>C1FlexGrid Column Editor</b> , or programmatically.	At design time, columns can be resized using the <b>Width</b> property in the Properties window or programmatically.
<b>Data Binding</b>	You can access the <b>Data Source Configuration Wizard</b> to bind to a data source either through the Properties window or the <b>C1FlexGrid Tasks</b> menu.	You can access the <b>Data Source Configuration Wizard</b> to bind to a data source through the Properties window.
	You can bind to a data source using any data provider supported by the .NET Framework.	When binding to a data source, you must use a data provider that is supported by the .NET Compact Framework. For example, you can use the .NET Framework Data Provider for SQL Server Mobile Edition. See <a href="#">Binding to a Data Source</a> for more information.
	The <b>DataSource</b> property can be set at design time or run time.	The <b>DataSource</b> property can be set at run time only.
<b>Image Support</b>	Use the <a href="#">SetCellImage</a> method to add an image to a cell. For example: <b>To write code in Visual Basic</b>	Use the <a href="#">SetCellImage</a> method to add an image to a cell <b>Image.FromFile</b> is not supported by the .NET Compact Framework, but you can load and set the image as in the following example: <b>Visual Basic</b>
	<pre>Visual Basic C1FlexGrid1.SetCellImage(1, 1, Image.FromFile(c:\myimage.bmp))</pre> <b>To write code in C#</b> <pre>C# c1FlexGrid1.SetCellImage(1, 1, Image.FromFile(@"c:\myimage.bmp"));</pre>	<pre>Visual Basic Dim myimage As New Bitmap("c:\myimage.bmp") C1FlexGrid1.SetCellImage(1, 1, myimage)</pre> <b>To write code in C#</b> <pre>C# { Bitmap myimage = new Bitmap("c:\myimage.bmp");</pre>

	C1FlexGrid (.NET)	C1FlexGrid (Mobile)
		<pre>c1FlexGrid1.SetCellImage(1, 1, myimage); }</pre> <p> <b>Note:</b>The <b>Image</b> class is unrelated to C1FlexGrid. Please see <a href="#">Microsoft's documentation</a> for more information.</p>
<b>Custom Merging</b>	You can use the <b>Custom</b> option in the <a href="#">AllowMerging</a> property. More information is available on custom merging in the <a href="#">Custom Merging</a> topic.	The <b>Custom</b> option is not available in the AllowMerging property, but custom merging can still be performed in the mobile version of C1FlexGrid. You can create a new class that derives from the C1FlexGrid and override the GetMergedRange virtual method, providing your own custom merging logic. See the <b>CustomMerge</b> samples provided with C1FlexGrid and on <a href="#">GrapeCity website</a> .

## Differences Between FlexGrid for WinForms and True DBGrid for WinForms

Many customers ask what the difference is between our grid components. While both are robust, easy-to-use grid controls that allow you to browse, edit, add, delete, and manipulate tabular data, there are several reasons why you may want to use one over the other.

Both components can be used in bound or unbound mode, but [C1FlexGrid](#) allows you to work more easily in unbound mode. With C1FlexGrid you can customize trees and take advantage of its cell merging capabilities. You can also derive from it to create customized grids.

**True DBGrid for WinForms** is better suited for data binding and, therefore, offers more features in that area, including split views and built-in hierarchical binding and grouping.

If you plan to use your grids mainly in bound mode and require advanced features such as splits and hierarchical views, **True DBGrid for WinForms** is the right choice. However, if you plan to work with your grids in unbound mode or need to customize the grid beyond what the object model offers, C1FlexGrid is a better choice.

If you have additional questions about **FlexGrid for WinForms** and **True DBGrid for WinForms**, please visit our Web site at <https://www.grapecity.com>.

## Key Features

Some of the key features of **FlexGrid for WinForms** that you may find useful include:

- **Code-free Development**

Set up columns and styles at design time with easy-to-use editors accessible from the SmartTag. The column editor allows you to insert and remove columns, reorder columns, adjust column widths, and more. Modify existing styles and add custom styles with the style editor. Completely manage your grid without writing any code!

- **Enhanced Cell Editing**

Use simple text editing, drop-down lists and combo lists, cell buttons, masks, and advanced data validation to control the editing process. For example, you can use an input mask to provide a template that automatically validates data as your users type, or choose to prevent users from editing specific columns altogether.

- **Integrated Printing**

Print your grid with a single statement! You have control over paper orientation, margins, and footer text, or you can show a dialog box to let your users select and set up the printer. Printing events allow you to control page breaks, add repeating header rows, or add custom elements to each page.

- **Hierarchical Styles**

View data the way that's best for you and your users with powerful properties and methods. For example, you can summarize data and add aggregate values with the [Subtotal](#) method or use the Tree property to display hierarchical views of the data.

- **Flexible Data Binding**

Use the grid in bound mode, where it displays data from any .NET data source, including ADO.NET and **DataObjects for .NET**, or in unbound mode, where the grid itself manages the data.

- **Enhance Your UI with Microsoft Office 2007 and 2010 Styling**

FlexGrid supports Visual Styles that mimic the styles available in Office 2007 and 2010, including Blue, Silver, and Black.

- **Child Tables for Displaying Hierarchical Data**

When FlexGrid is bound to a hierarchical data source, each master record can be expanded/collapsed to show/hide the details in child grids, which may in turn contain more details. The result is a "data tree" similar to the type of grid presented by Microsoft Access when displaying hierarchical data. This is accomplished by deriving a control (C1FlexDataTree) from the [C1FlexGrid](#) control. When bound, the control detects subordinate data sources and creates additional instances of itself to display child tables.

- **Make it a Tree**

By setting the [IsNode](#) property for certain rows you can transform FlexGrid into a TreeView. You get a TreeView that has all the features of a rich datagrid.

- **Display Images and Data in Cells**

Each grid cell can display images in addition to data. You can even bind grid columns to image lists, an easy and efficient way to display database information graphically.

- **Add Special Drawing Effects**

Make the grid look the way you want it to by painting special effects such as lines, bitmaps, and icons in appropriate grid cells. You can also scale images and add transparency.

- **Assign Field Names to Columns**

Refer to columns by name instead of position, if desired. Column keys are automatically assigned to field names when the grid is data-bound, or you may assign them with code. You can later refer to a column using a `ColIndex(ColKey)` syntax, which will retrieve the column you want even if your user has moved it to a different position on the grid.

- **Merged Cells**

Merge contiguous like-valued cells, making them span multiple rows or columns to enhance appearance or clarity.

- **Range Aggregates**

Calculate totals, averages, and other statistics for ranges of cells with a single statement!

- **Multiple File Formats for Saving/Loading Data**

Load from and save grid contents to text, .xls, .xlsx (OpenXml format) files, or Open XML files with other extensions. You can also load grid data from a database using `DataReader` objects.

- **Built-in Data Filtering**

Allow users to apply filters to each column on the grid by setting the [AllowFiltering](#) property.

- **Grouping**

Allow users to group the data in FlexGrid by combining rows based on column values and perform grouping using any of the two ways, using `FlexGridColumnPanel` or through code.

- **Instant Search Panel**

Search the entire grid in one go and locate entries among millions of records instantly by using FlexGrid's Instant Search Panel.

- **Show Sparklines**

FlexGrid for WinForms supports displaying the Line, Column and Winloss sparklines in FlexGrid columns which have data of type array, list or `ObservableCollection`. Apart from this, it also supports extended features of sparkline such as styling sparklines and displaying X axis, markers etc.

## Feature Comparison Matrix

Explore all the features offered by FlexGrid, TrueDBGrid, and MS DataGridView. You can [download the matrix in PDF](#).

### Data Binding

Features	FlexGrid	TrueDBGrid	MS DataGridView
Binding Data Sources	✓	✓	✓
Binding Data Sources with hierarchical data relations	With custom code	✓	
Unbound data storage and manipulations	✓	✓	✓

### Data Presentation

Features	FlexGrid	TrueDBGrid	MS DataGridView
Hierarchical Styles	With custom code	✓	
TreeView-like Styles	✓	✓	
MultiLine Data View		✓	
Grouping	✓	✓	
Built-in Drag And Drop Grouping	✓	✓	
Horizontal and Vertical Interactive Splits		✓	
Child grids within master grid support	✓	✓	
Drop-Down Objects Support	✓	✓	
Drop-Down Multicolumn Object	✓	✓	

support			
Drop-Down Multicolumn Bindable and Sortable Object support		✓	

## Data Exchange

Features	FlexGrid	TrueDBGrid	MS DataGridView
Export Data (Delimited Text, XLS and XLSX)	✓	✓	
Export Data in other formats (PDF, HTML, RTF, JPG and other)		✓	
Load data from Excel files	✓	✓	
Enhanced Printing data from grid and Print Preview support	✓	✓	

## Cells Manipulations

Features	FlexGrid	TrueDBGrid	MS DataGridView
In-Cell Objects	✓	✓	✓
Enhanced Cell Editing with Custom Editors	✓	✓	
Merging Cells and Rows	✓	✓	
Customize	✓		



Cells Merging			
Drag And Drop Columns and rows	✓	✓	
Automatic Cell Sizing	✓	✓	✓
Fixed, Non-scrolling Columns	✓	✓	✓
Excel-style Cell Selection	✓	✓	
Customize each Cell Rendering	✓		
Zooming Cells	✓	✓	
Run-Time CellTips	✓	✓	
Data Manipulations with Cell Ranges	✓		

## Layout and Styling

Features	FlexGrid	TrueDBGrid	MS DataGridView
Visual Styles Support	✓	✓	✓
Dynamic Support of 38 Decoration Themes	✓	✓	
Office 2007 and 2010 Styling	✓	✓	
Alternating Row Colors	✓	✓	✓
Customizable Cell Border Style	✓	✓	
Add Special Drawing	✓	✓	

Effects			
Data-Sensitive Display	✓	✓	

## Input and Navigation

Features	FlexGrid	TrueDBGrid	MS DataGridView
Input Masking	✓	✓	✓
Simplify Data Input	✓	✓	
Automatic Data Translation	✓	✓	
Spell Checking support	✓	✓	
Customizable Keyboard Navigation and keys behavior	✓	✓	
Right-To-Left navigation	✓	✓	✓
Touch Support	✓	✓	✓
Clipboard support	✓	✓	✓
Rich Scrolling Capabilities	✓	✓	

## Data Manipulations

Features	FlexGrid	TrueDBGrid	MS DataGridView
Sorting	✓	✓	✓
Multi-column Sorting	✓		
Built-in Data Filtering	✓	✓	
Extended and Conditional Filtering	✓	✓	
Customizable filter form	✓		
Additional Filter Bar row		✓	
Multilanguage Filtering	✓	✓	

AutoSearch	✓		
Range Aggregates	✓	✓	

## Localization

Features	FlexGrid	TrueDBGrid	MS DataGridView
Right-To-Left support	✓	✓	✓
.Net localization support	✓	✓	✓
Regional settings support	✓	✓	✓

## Other Features

Features	FlexGrid	TrueDBGrid	MS DataGridView
Optimize performance for displaying large amount of data in bound and unbound mode	✓	✓	
Just-In-Time Data Loading	✓	✓	✓
Server-side Data Virtualization with C1DataSource	✓	✓	✓
Automatic Lookup Columns with C1DataSource	✓		✓
Design-time extended support	✓	✓	
Assembly size	1508 K	2108 K	Part of System.Windows.Forms
Pricing	\$	\$	\$

## FlexGrid for WinForms Quick Start

In this section you will learn how to use the basic [C1FlexGrid](#) functionality to create a simple grid application. This section is not supposed to be a comprehensive tutorial on all features of C1FlexGrid, but rather provide a quick start and highlight some general approaches to using the product. For more in-depth tutorials, see the [FlexGrid for WinForms Tutorials](#). Visual Studio 2010 was used in this example; the steps may be slightly different in other versions of Visual Studio.

## Step 1 of 3: Creating the FlexGrid for WinForms Application

The following steps will walk you through creating a simple grid application.

1. Create a new project.
2. Add a [C1FlexGrid](#) control to the form.
3. Open the **C1FlexGrid Tasks** menu. For more information on accessing the **C1FlexGrid Tasks** menu, see [C1FlexGrid Tasks Menu](#).
4. In the **C1FlexGrid Tasks** menu, click **Dock in parent container**. This sets the grid's **Dock** property to **Fill** so the grid will fill the form.
5. Run the program. A simple grid application will appear.

Congratulations! You have successfully created a simple grid application. In the next topic, you will learn how to bind the C1FlexGrid control to a data source.

## Step 2 of 3: Binding C1FlexGrid to a Data Source

The following steps will walk you through binding a data source to the grid application you created in the [Step 1 of 3: Creating the FlexGrid for WinForms Application](#) topic.

1. Open the **C1FlexGrid Tasks** menu. For more information on accessing the **C1FlexGrid Tasks** menu, see [C1FlexGrid Tasks Menu](#).
2. In the **C1FlexGrid Tasks** menu, click the **Choose Data Source** drop-down arrow and select the **Add Project Data Source** link from the drop-down box.
3. The **Data Source Configuration Wizard** appears. Leave the default setting, **Database**, selected on the **Choose a Data Source Type** page, and click **Next**.
4. On the **Choose a Database Model** page, leave **Dataset** selected and click **Next**.
5. Click the **New Connection** button to create a new connection or choose one from the drop-down list. When you click **New Connection**, the **Add Connection** dialog box appears.
6. Leave **Microsoft Access Database File** as the **Data source**.
7. Click the **Browse** button under **Database file name**. In the **Select Microsoft Access Database File** dialog box, browse to the C1NWind.mdb database in the **Documents\ComponentOne Samples\Common** directory. Select the C1NWind.mdb file and click **Open**.
8. In the **Add Connection** dialog box, click the **Test Connection** button to make sure that you have successfully connected to the database or server and click **OK**.
9. Click **OK** again to close the **Add Connection** dialog box.
10. Click the **Next** button to continue. A dialog box will appear asking if you would like to add the data file to your project and modify the connection string. Since it is not necessary to copy the database to your project, click **No**.
11. Save the connection string in the application configuration file by checking the **Yes, save the connection as** box and entering a name. Click the **Next** button to continue.
12. On the **Choose Your Database Objects** page, expand the **Tables** node, and select the **Products** table. Enter **ProductsDS** in the **DataSet name** box and click **Finish** to exit the wizard.
13. A DataSet and connection string are added to your project. Additionally, Visual Studio automatically creates

the following code to fill the DataSet:

### To write code in Visual Basic

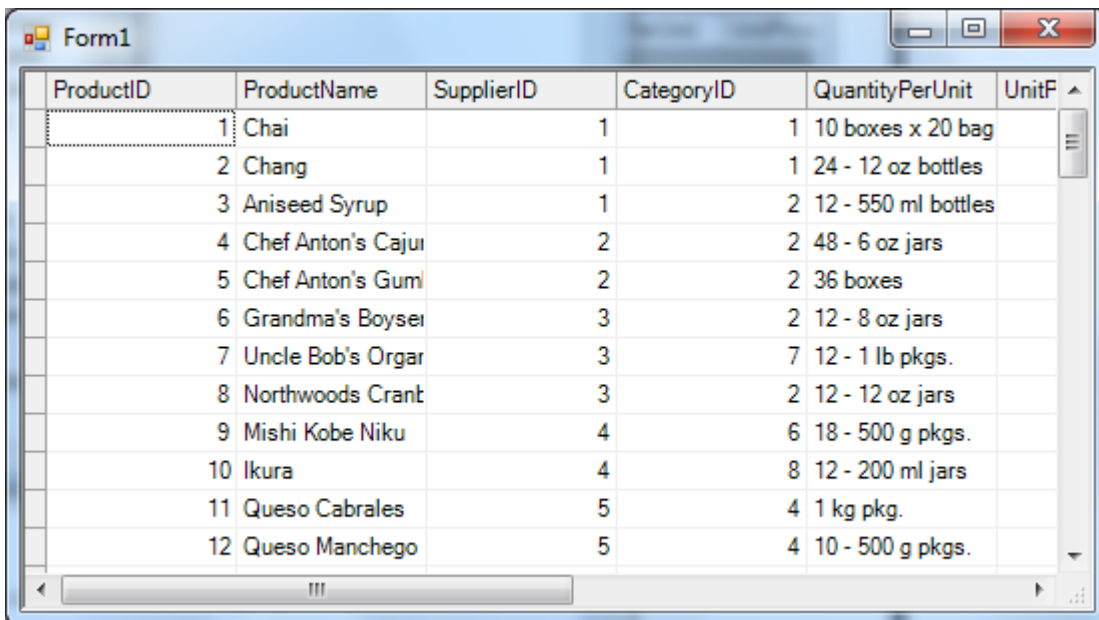
```
Visual Basic  
Me.ProductsTableAdapter.Fill(Me.ProductsDS.Products)
```

### To write code in C#

```
C#  
this.productsTableAdapter.Fill(this.productsDS.Products);
```

## Run the program and observe the following:

Notice that the data from the **Products** table is reflected in the grid.



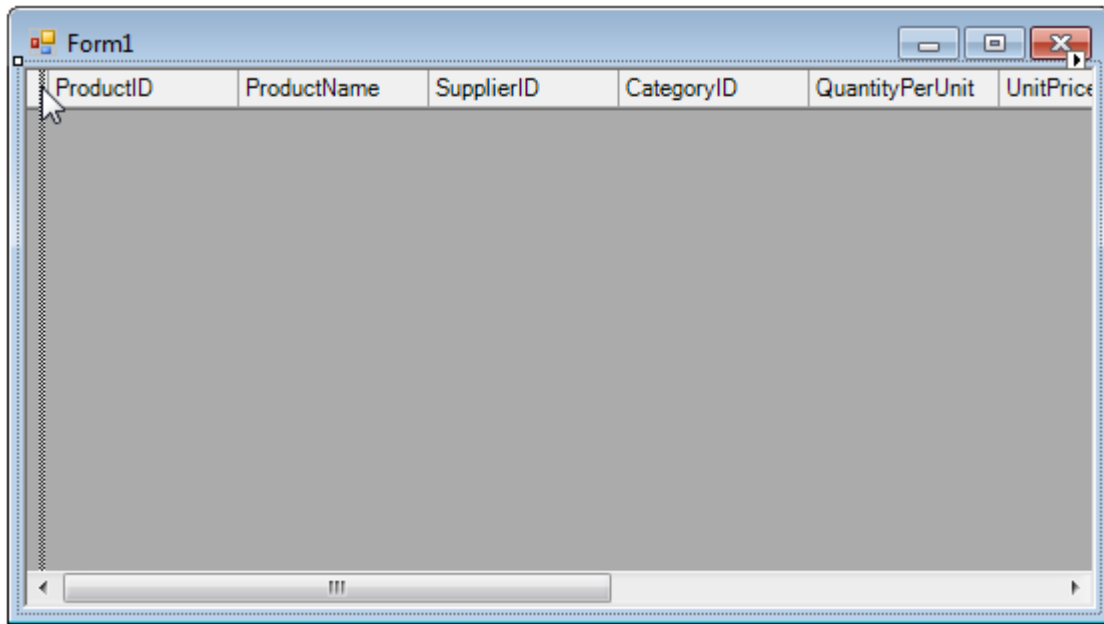
ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitF
1	Chai	1	1	10 boxes x 20 bag	
2	Chang	1	1	24 - 12 oz bottles	
3	Aniseed Syrup	1	2	12 - 550 ml bottles	
4	Chef Anton's Cajun	2	2	48 - 6 oz jars	
5	Chef Anton's Gum	2	2	36 boxes	
6	Grandma's Boyse	3	2	12 - 8 oz jars	
7	Uncle Bob's Orgar	3	7	12 - 1 lb pkgs.	
8	Northwoods Crant	3	2	12 - 12 oz jars	
9	Mishi Kobe Niku	4	6	18 - 500 g pkgs.	
10	Ikura	4	8	12 - 200 ml jars	
11	Queso Cabrales	5	4	1 kg pkg.	
12	Queso Manchego	5	4	10 - 500 g pkgs.	

Congratulations! You have successfully bound the grid application to a data source. In the next topic, you will learn how to customize format strings, Visual Styles, and built-in styles.

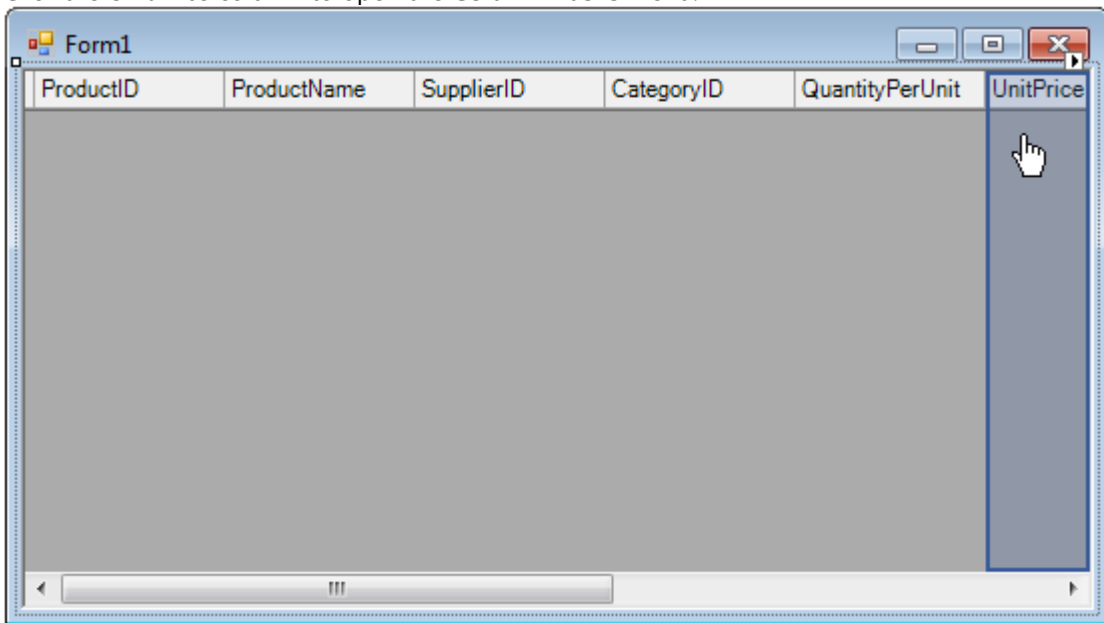
## Step 3 of 3: Customizing C1FlexGrid Settings

The following steps will walk you through setting a format string, Visual Style, and built-in styles for the grid.

1. Resize the first column in the grid by dragging the horizontal double arrow that appears at the right edge of the first column's header to the left until the *UnitPrice* column is visible:



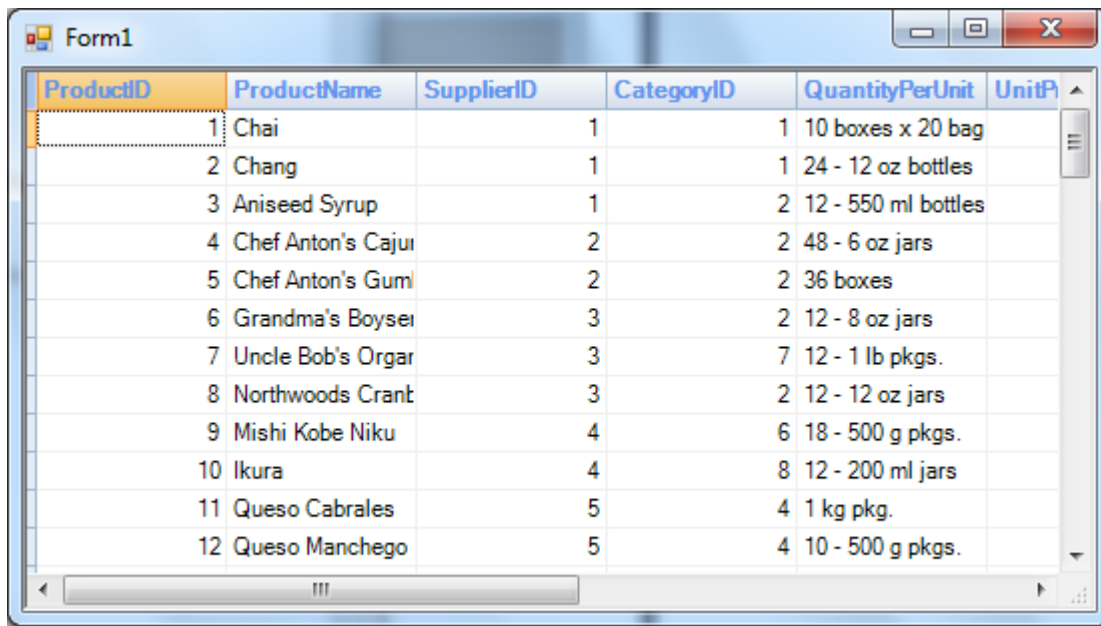
2. Click the *UnitPrice* column to open the **Column Tasks** menu.



3. Click the **ellipsis** button next to the **Format String** text box to open the **Format String** dialog box.
4. In the **Format String** dialog box, select **Currency** under **Format type** and click **OK**.
5. In the Properties window, locate the **VisualStyle** property and set it to **Office2007Blue**.
6. Open the **C1FlexGrid Tasks** menu. The **Column Tasks** menu will appear since we last edited a column using the **Tasks** menu. Select **C1FlexGrid Tasks** to return to the **C1FlexGrid Tasks** menu.
7. On the **C1FlexGrid Tasks** menu, select **Styles** to open the **C1FlexGrid Style Editor**.
8. In the **C1FlexGrid Style Editor** under **Built-In Styles**, select **Fixed**.
9. Expand the **Font** node in the right pane and set the **Bold** property to **True**.
10. Set the **ForeColor** property to **CornflowerBlue** on the **Web** tab and click **OK** to close the dialog box.

## Run the program and observe the following:

The grid application displays a the **Products** table using a format string, Visual Style, and built-in styles.



The screenshot shows a Windows Forms application window titled "Form1". Inside the window is a FlexGrid control displaying a table of product data. The grid has six columns: ProductID, ProductName, SupplierID, CategoryID, QuantityPerUnit, and UnitP. The data is as follows:

ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitP
1	Chai		1	10 boxes x 20 bag	
2	Chang		1	24 - 12 oz bottles	
3	Aniseed Syrup		1	2 12 - 550 ml bottles	
4	Chef Anton's Cajun	2		48 - 6 oz jars	
5	Chef Anton's Gumbo	2		36 boxes	
6	Grandma's Boysea	3		12 - 8 oz jars	
7	Uncle Bob's Organ	3		7 12 - 1 lb pkgs.	
8	Northwoods Cranb	3		2 12 - 12 oz jars	
9	Mishi Kobe Niku	4		6 18 - 500 g pkgs.	
10	Ikura	4		8 12 - 200 ml jars	
11	Queso Cabrales	5		4 1 kg pkg.	
12	Queso Manchego	5		4 10 - 500 g pkgs.	

You have successfully set the format string, Visual Style, and built-in styles for the grid. This concludes the Quick Start.

## Design-Time Support


You can easily configure **FlexGrid for WinForms** at design time using the property grid, menus, and designers in Visual Studio. The following sections describe how to use C1FlexGrid's design-time environment to configure the [C1FlexGrid](#) control.

## C1FlexGrid Editors

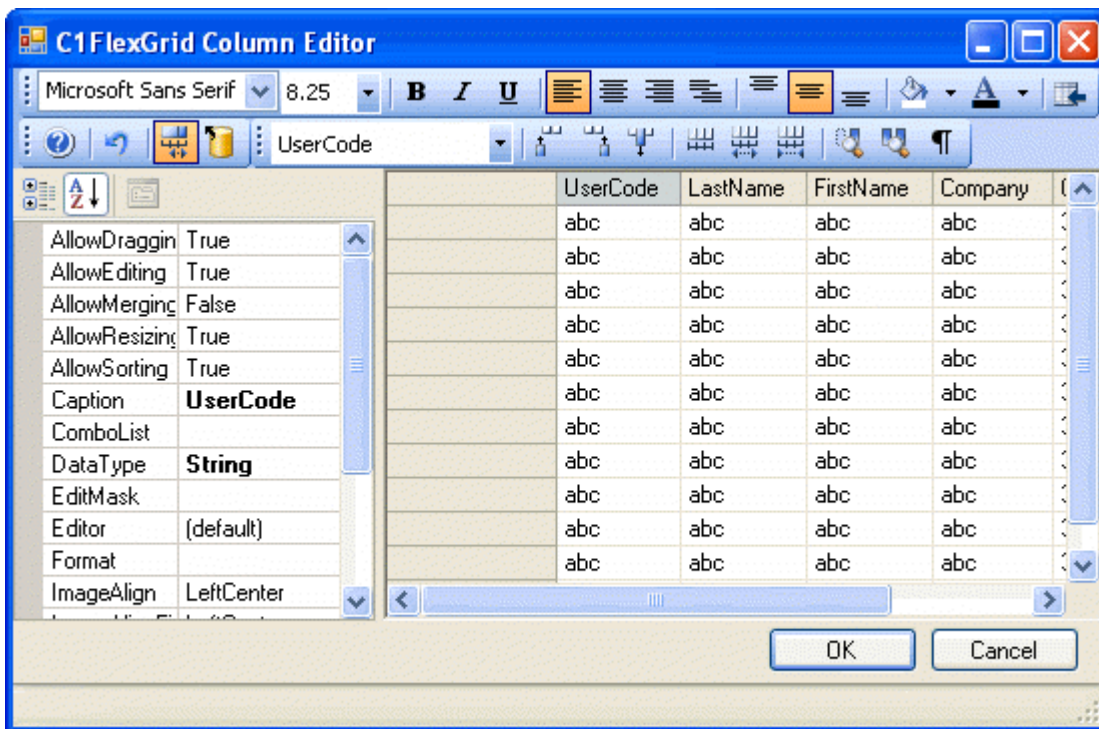
There two are design-time editors that allow you to control the layout and appearance of [C1FlexGrid](#), **C1FlexGrid Column Editor** and **C1FlexGrid Style Editor**. Additionally, **Caption Style** and **Column Style** are two design-time editors that allow you to change the appearance of a specific caption or column.

## C1FlexGrid Column Editor

If you prefer, you can set up the grid columns at design time instead of writing code to do it. The designer can be accessed in one of three ways:

- Select the grid in Design view, go to the Properties window and click the **ellipsis** button (...) next to the **Cols** property.
- Right-click the control and select **Designer** from the context menu.
- Click on the smart tag (  ) in the upper right corner of the grid and select **Designer** from the **C1FlexGrid Tasks** menu.

This will bring up the **Column Editor** shown below:

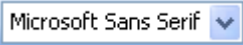
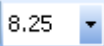












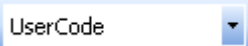







In bound mode, the editor can be used to select which fields in the [DataSource](#) should be displayed, their order, column captions, widths, and alignment. In unbound mode, the editor is also used to select column data types.



The editor allows you to perform the following actions:

- **Reorder Columns:** You can move columns to new positions by dragging them by the header cells with the mouse.
- **Adjust Column Widths:** You can adjust column widths with the mouse, by dragging the right edge of the header cells with the mouse. You can also select multiple columns by SHIFT-clicking the header cells, and then set all column widths at once using the property window. Setting the column width to -1 restores the default width.
- **Set Column Properties:** Whenever one or more columns are selected, you can see and edit their properties in the property grid on the left of the editor.
- **Insert or Remove Columns:** Use the toolbar to insert columns before or after the selection (useful mostly in unbound mode), or to remove columns.
- **Use the Toolbar to Perform Common Tasks:** The table below describes the function of the buttons on the toolbar:

Button	Description
	<b>Font:</b> Sets the selected font in the drop-down list.
	<b>Font Size:</b> Sets the selected font size in the drop-down list.
	<b>Font Formatting:</b> Applies bold, italics or underline to the font.
	<b>Justification:</b> Sets the font alignment to left, center, right or general.
	<b>Alignment:</b> Aligns column content to the top, center, or bottom. These buttons only affect the scrollable area of the grid. To set the alignment for the header columns, select the columns and set the <b>TextAlignFixed</b> property.
	<b>BackColor:</b> Sets the backcolor of the selected column.
	<b>ForeColor:</b> Sets the forecolor of the selected column.
	<b>Apply to Fixed Rows:</b> Applies the settings to fixed rows.
	<b>Help:</b> Displays or hides the description for the selected property.
	<b>Undo:</b> Cancels all changes and reverts the grid columns to their original state.
	<b>AutoResize:</b> Determines whether the grid should automatically resize all columns to fit their contents when the grid is bound to a data source.
	<b>Reload from Datasource:</b> Resets all columns with information

Button	Description
	from the current <b>DataSource</b> . This button is useful when the grid is bound to a data source and you want to start editing from scratch. The button is disabled when the grid is not bound to a data source.
	<b>Selected Column:</b> Selects the current column from the drop-down list.
	<b>Insert Column:</b> Inserts columns to the left or right of the selection.
	<b>Delete Column:</b> Removes the selected column.
	<b>Column Width:</b> Sets the column width of all selected columns to the same width, wider or narrower.
	<b>Toggle Visibility:</b> Displays or hides a column.
	<b>Unhide All Columns:</b> Makes all columns visible.
	<b>Show Hidden Columns:</b> Shows hidden columns. If you change the Visible property of a column to False, it will be hidden, and therefore you won't be able to select it with the mouse. Use this button to show all hidden columns so you can select and edit them.
	Determines whether the properties for the selected columns should be displayed in categorized or alphabetical order.

## C1FlexGrid Style Editor

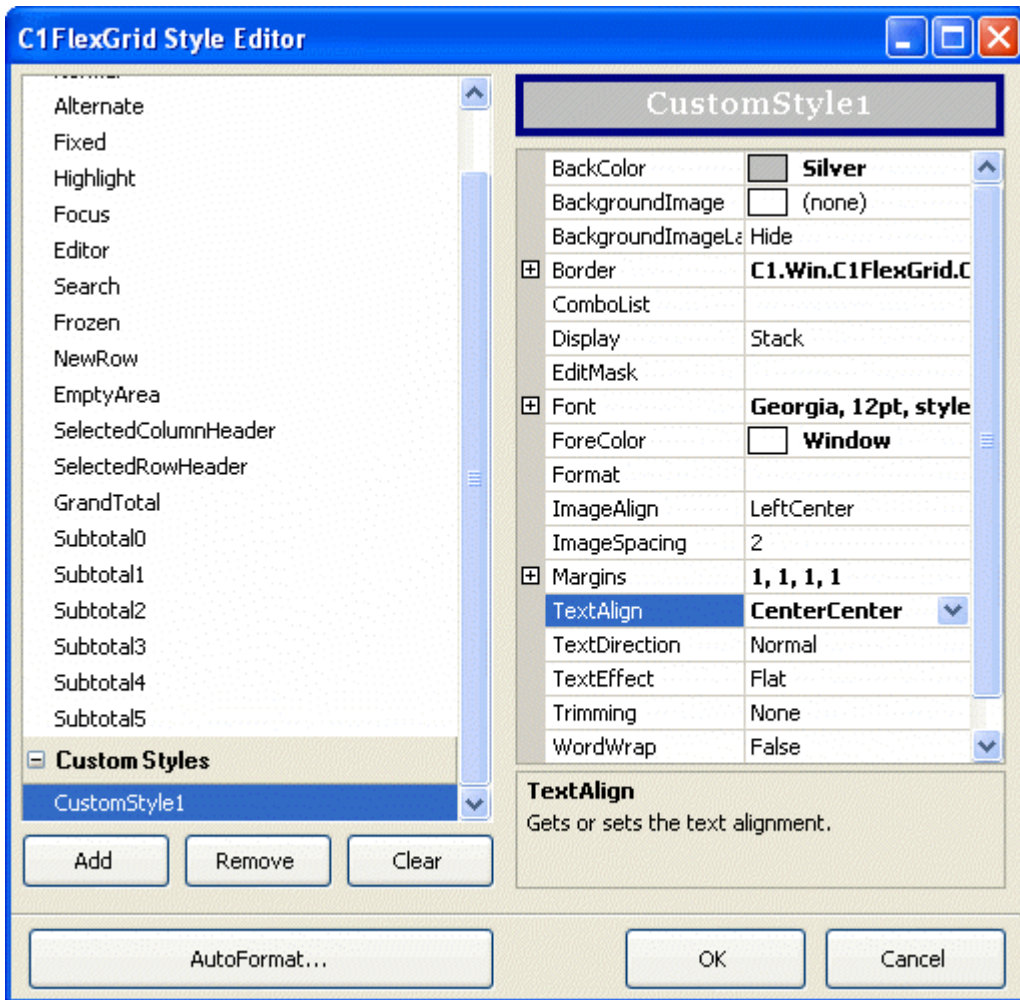
If you prefer, you can set up styles at design time instead of writing code to do it. The **C1FlexGrid Style Editor** can be accessed in one of three ways:

- Select the grid, go to the Properties window and click the **ellipsis** button next to the [Styles](#) property.
- Right-click the grid and select **Styles** from the context menu.
- Click on the smart tag (📌) in the upper right corner of the grid and select **Styles** from the **C1FlexGrid Tasks** menu.

The grid will display the **C1FlexGrid Style Editor** dialog box.

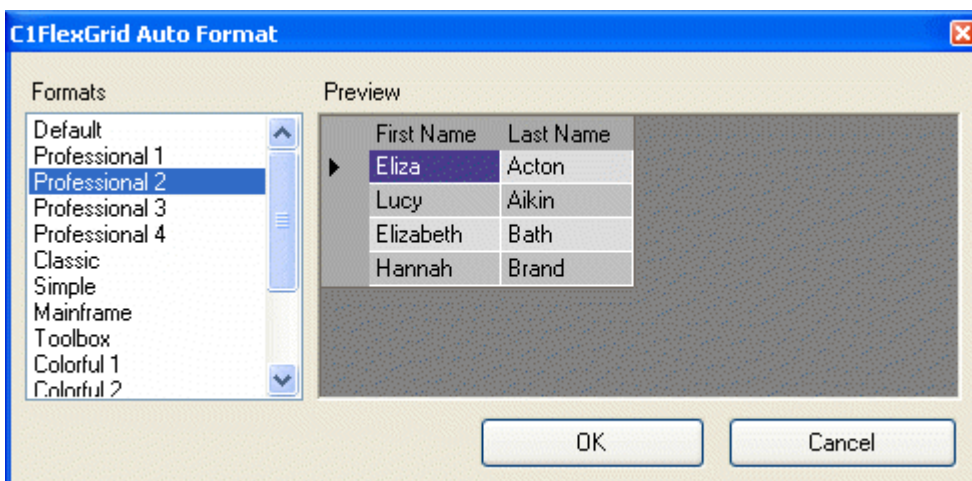
The style editor lets you modify existing styles and add new custom ones, which may later be assigned to cells, rows, and columns.

Use the **Add** button to add a custom style. You can see how your new style appears in the preview area above the style properties.



The **Remove** button removes the selected custom style. You can rename custom styles by selecting them on the list and typing the new name. The **Clear** button removes all custom styles and restores the built-in styles to their default values.

The **AutoFormat** button brings up a secondary dialog box that allows you to select a complete set of predefined styles. Here's what the **AutoFormat** dialog box looks like:



## Caption Style and Column Style

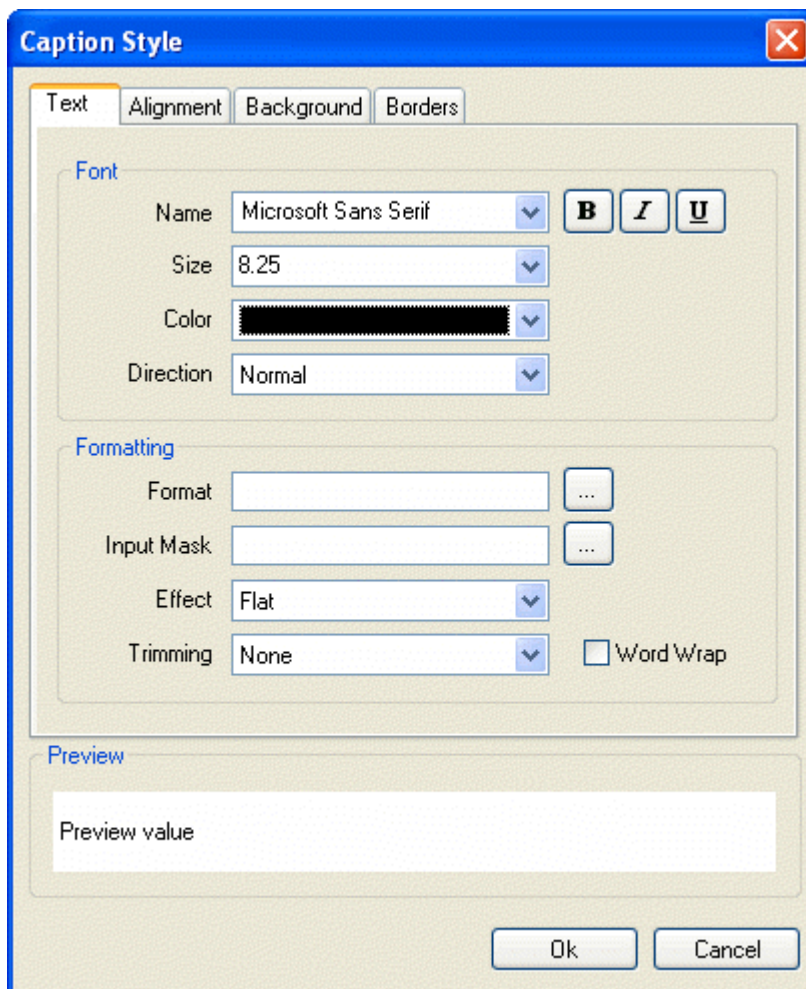
The **Caption Style** editor and the **Column Style** editor for a selected column allow you to specify the properties for the caption text and column text, respectively, as well as alignment, background, and borders.

The **Caption Style** and **Column Style** editors can only be accessed in the **Column Tasks** menu. For more information on the **Column Tasks** menu, see [Column Tasks Menu](#).

The **Caption Style** and **Column Style** editors have four tabs: **Text**, **Alignment**, **Background**, and **Borders**. The **Preview** area allows you to view your settings before applying them to the grid.

### Text

The **Text** tab sets the font and formatting of the caption.



The following options are available in the **Font** area:

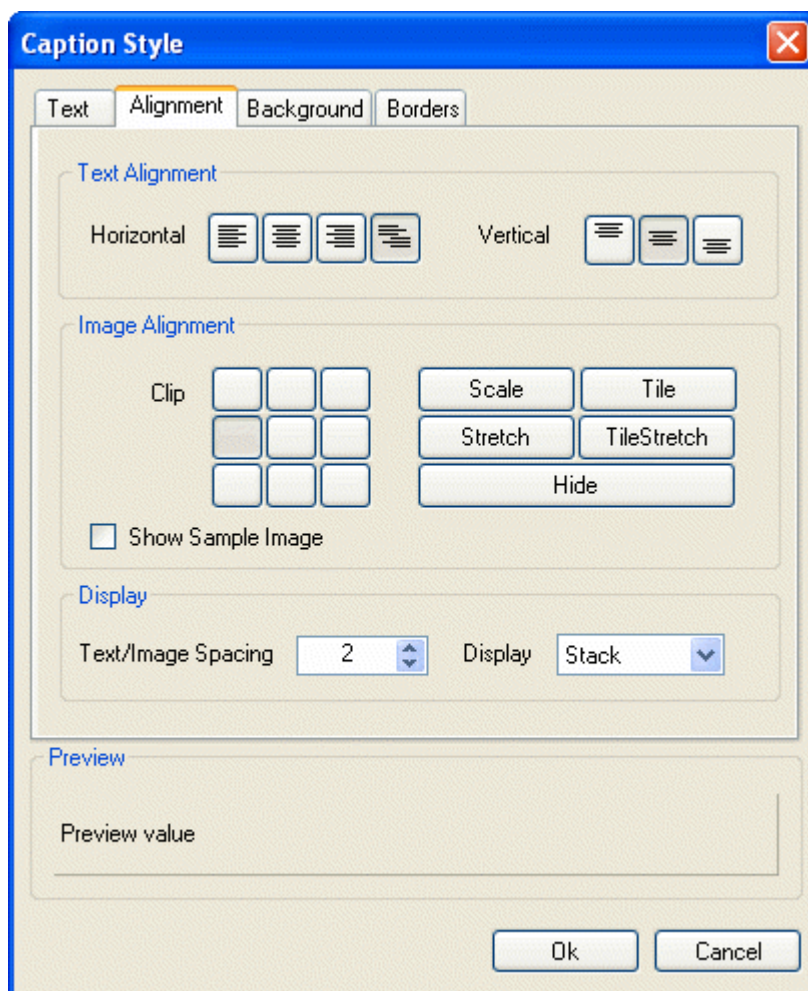
- **Name:** Choose a font name.
- **Size:** Choose a font size.
- **Color:** Choose a font color.
- **Direction:** Choose from **Normal**, **Up** or **Down**.
- **Font Effects:** Use the buttons to toggle bold, italic, and underline on or off.

The following options are available in the **Formatting** area:

- **Format:** Click the **ellipsis** button to open the **Format String** dialog box. For more details on the **Format String** dialog box, see [Cell Content](#).
- **Input Mask:** Click the **ellipsis** button to open the **Input Mask** dialog box. For more details on the **Input Mask** dialog box, see [Masks](#).
- **Effect:** Choose from **Flat**, **Raised**, or **Inset**. For more details on the text effect options, see the [TextEffectEnum Enumeration](#).
- **Trimming:** Choose from **None**, **Character**, **Word**, **EllipsisCharacter**, **EllipsisWord**, or **EllipsisPath** to set how long strings are trimmed to fit the cell.
- **Word Wrap:** Check the box to enable word wrapping for the caption.

## Alignment

The **Alignment** tab sets the alignment of both text and images in the caption.



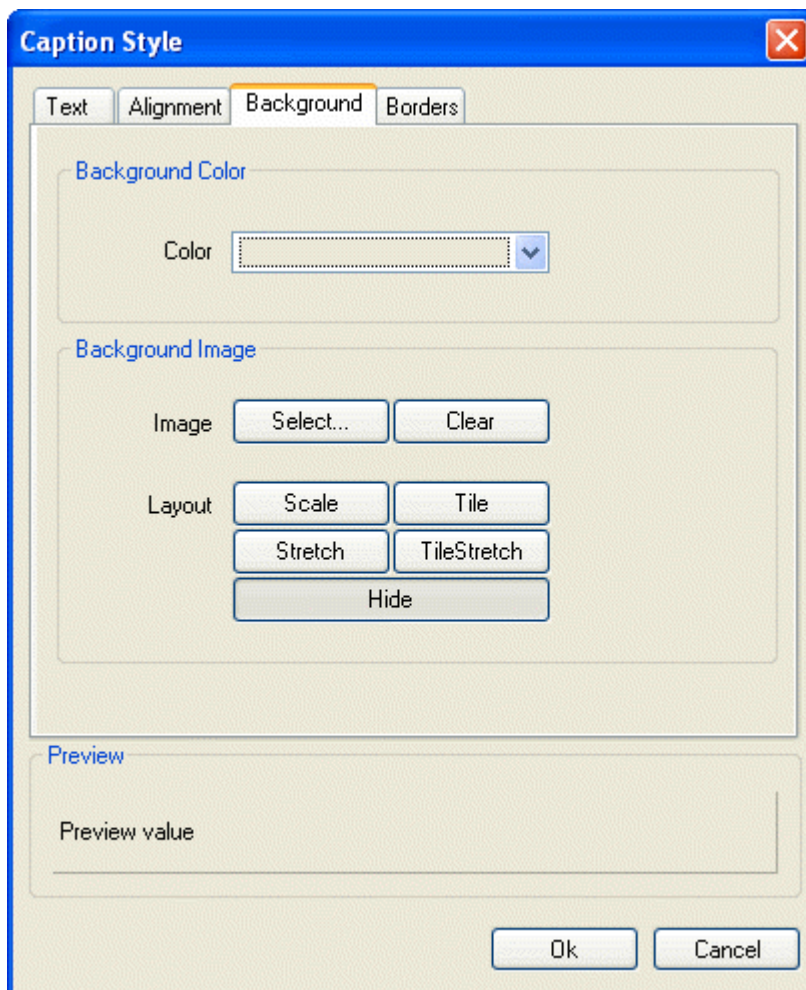
The following options are available in the **Text Alignment** area:

- **Horizontal:** Click the buttons to toggle between **Align Left**, **Align Center**, **Align Right**, and **Align General**.
- **Vertical:** Click the buttons to toggle between **Align Top**, **Align Middle**, and **Align Bottom**.
- The following options are available in the **Image Alignment** area:
- **Clip:** Click the buttons to toggle between image alignment in the cell or **Scale**, **Tile**, **Stretch**, **TileStretch**, or **Hide** the image. For more details on the image alignment options, see the [ImageAlignEnum Enumeration](#).

- **Show Sample Image:** Check the **Show Sample Image** box to display a sample image in the **Preview** area.
- The following options are available in the **Display** area:
- **Text/Image Spacing:** Increase or decrease this value to increase or decrease the amount of space between the text and the image.
- **Display:** Choose from **TextOnly**, **ImageOnly**, **Overlay**, **Stack**, or **None**. For more details on the display options, see the [DisplayEnum Enumeration](#).

## Background

The **Background** tab sets the background color and background image.



The following option is available in the **Background Color** area:

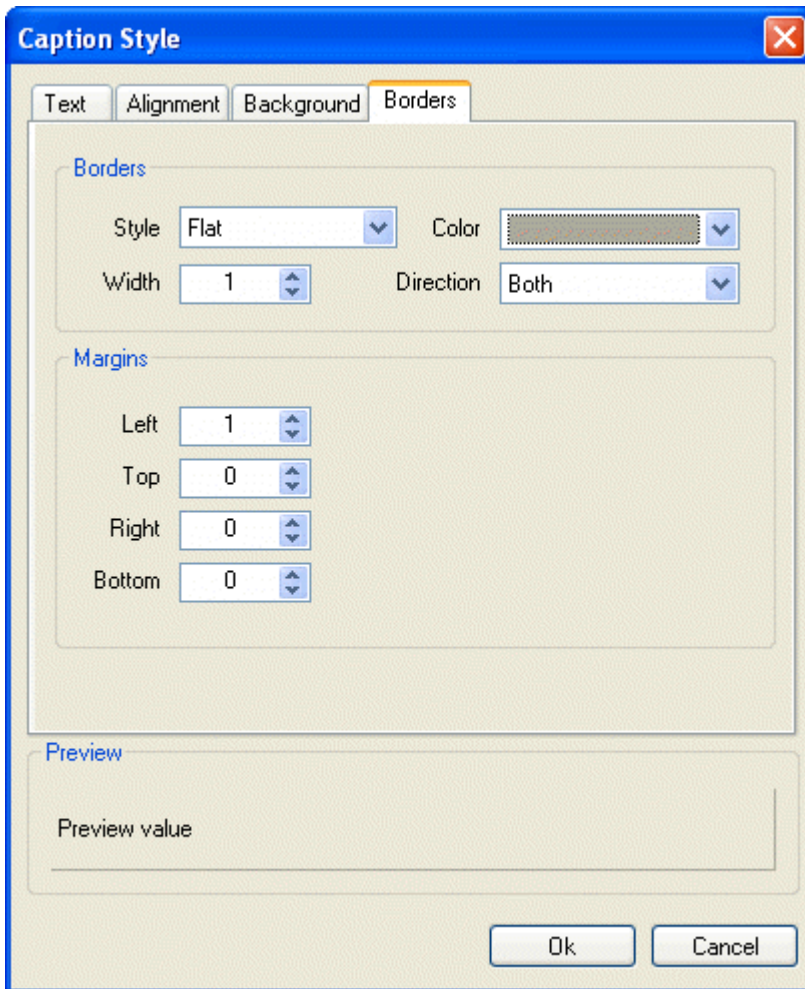
- **Color:** Choose a color for the background of the cell.

The following options are available in the **Background Image** area:

- **Image:** Click the **Select** button to select an image or the **Clear** button to remove an image.
- **Layout:** Toggle between **Scale**, **Tile**, **Stretch**, **TileStretch**, or **Hide**.

## Borders

The **Borders** tab sets the borders and margins.




The following options are available in the **Borders** area:

- **Style:** Choose from **None**, **Flat**, **Double**, **Raised**, **Inset**, **Groove**, **Fillet**, or **Dotted**. For details on the different border style options, see the [C1.Win.C1FlexGrid.BorderStyleEnum Enumeration](#).
- **Width:** Increase or decrease the value to increase or decrease the width of the border.
- **Color:** Choose a color for the border.
- **Direction:** Choose from **Both**, **Horizontal**, or **Vertical**. For details on the different border direction options, see the [BorderDirEnum Enumeration](#).

The following options are available in the **Margins** area:

- **Left:** Increase or decrease the value to increase or decrease the left margin.
- **Top:** Increase or decrease the value to increase or decrease the top margin.
- **Right:** Increase or decrease the value to increase or decrease the right margin.
- **Bottom:** Increase or decrease the value to increase or decrease the bottom margin.


## C1FlexGrid Smart Tag

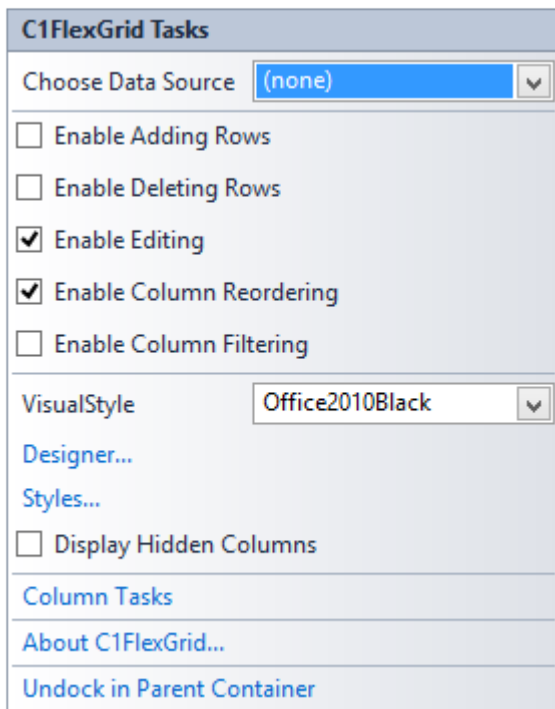
A smart tag (  ) represents a short-cut **Tasks** menu that provides the most commonly used properties in each control. There are two **Tasks** menus available through the smart tag in [C1FlexGrid](#): [C1FlexGrid Tasks Menu](#) and [Column Tasks](#)

Menu.

## C1FlexGrid Tasks Menu

In the **C1FlexGrid Tasks** menu, you can quickly access the **C1FlexGrid Column Editor** and the **C1FlexGrid Style Editor**, as well as set the following properties: [AllowAddNew](#), [AllowDelete](#), [AllowEditing](#), and [AllowDragging](#).

To access the **C1FlexGrid Tasks** menu, click on the smart tag (  ) in the upper right corner of the grid. This will open the **C1FlexGrid Tasks** menu.



### Choose Data Source

Clicking the drop-down arrow in the **Choose Data Source** box opens a list of available data sources and allows you to add a new data source. To add a new data source to the project, click **Add Project Data Source** to open the **Data Source Configuration Wizard**.

For more information on how to add a new data source to the project, see [Binding to a Data Source](#).

### Enable Adding Rows

Selecting the **Enable Adding Rows** check box sets the `AllowAddNew` property to **True**, and allows adding new rows to the grid. The default is unchecked.

### Enable Deleting Rows

Selecting the **Enable Deleting Rows** check box sets the `AllowDelete` property to **True**, and allows deleting rows in the grid. The default is unchecked.

### Enable Editing



Selecting the **Enable Editing** check box sets the AllowEditing property to **True**, and allows editing in the grid. The default is checked.

## Enable Column Reordering

Selecting the **Enable Column Reordering** check box sets the AllowDragging property to **Columns**, and allows dragging of columns in the grid. The default is checked.

## Enable Column Filtering

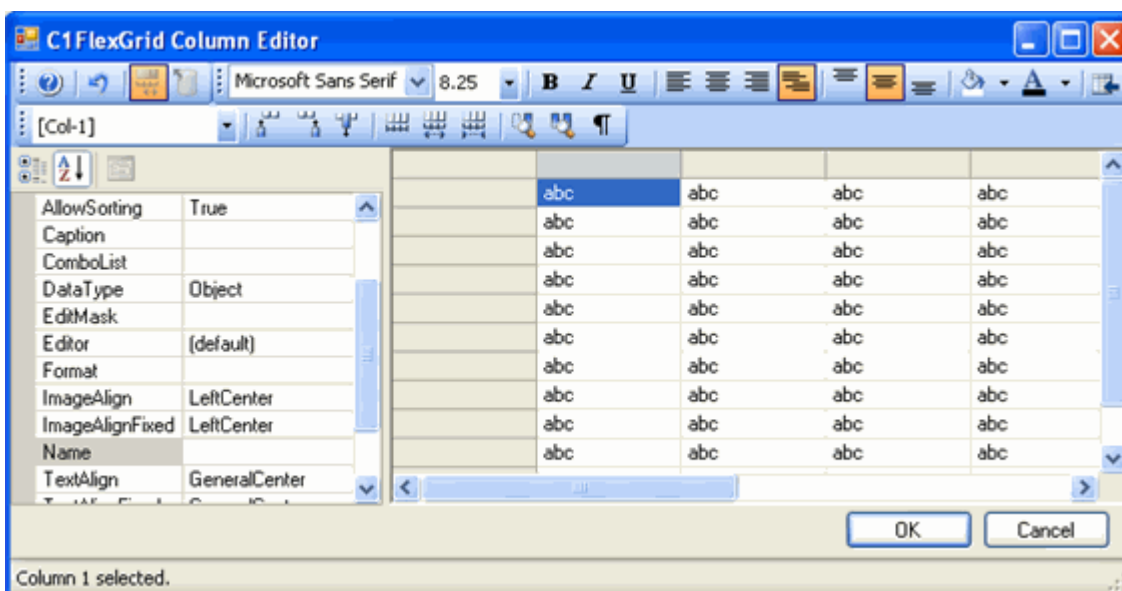
Selecting the **Enable Column Filtering** check box sets the AllowFiltering property to **Columns**, and allows filtering of columns in the grid. The options available for filtering are listed in the drop-down - Default, ByValue, ByCondition, and Custom. Condition filters allow you to specify conditions such as 'value > 10'. Value filters allow you to select values that should be displayed from a list of values present in the data source. Custom filters allow you to define your own filters through code.

## Display Hidden Columns

Selecting the **Display Hidden Columns** check box sets the Visible property to **True** for hidden columns and displays them. The default is unchecked.

## Designer

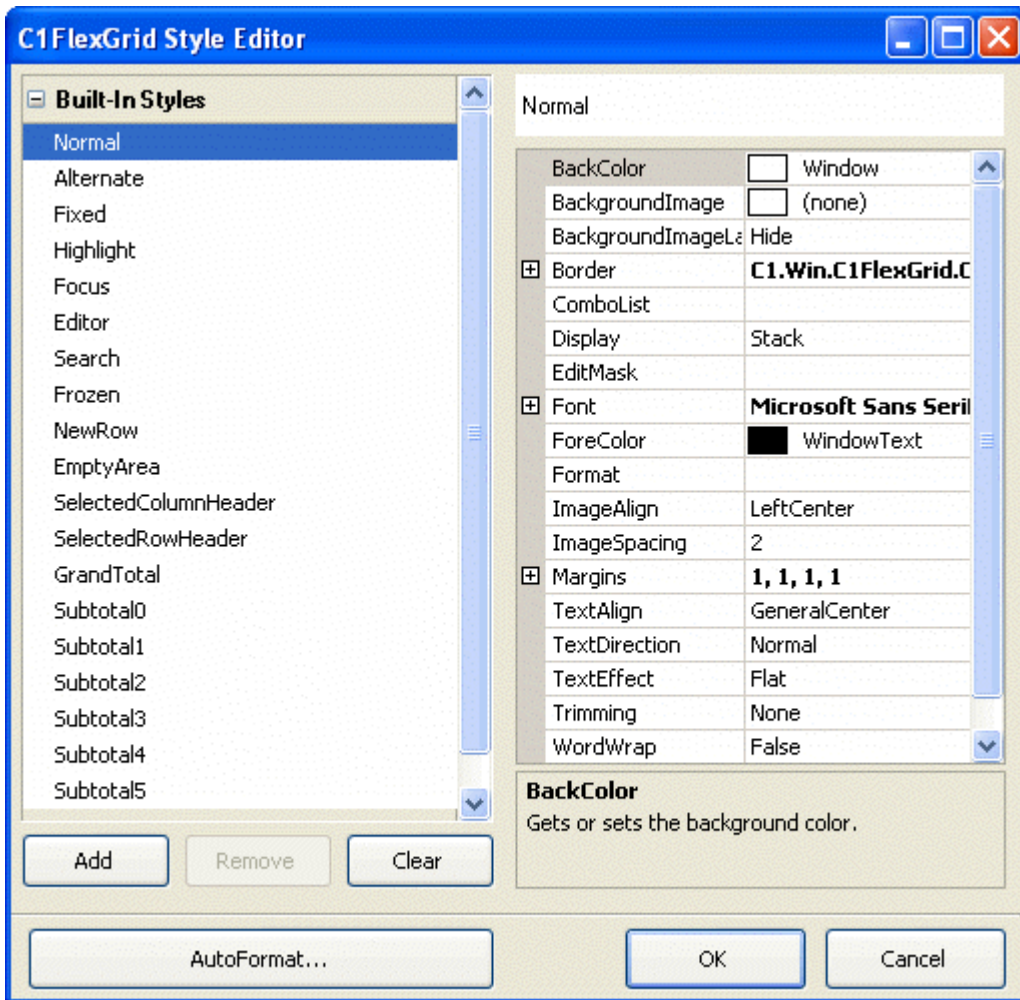
Clicking **Designer** opens the **C1FlexGrid Column Editor**.



For more information on how to edit columns with the **C1FlexGrid Column Editor**, see [C1FlexGrid Column Editor](#).

## Styles

Clicking **Styles** opens the **C1FlexGrid Style Editor**.



For more information on how to customize cell appearance with the **C1FlexGrid Style Editor**, see [C1FlexGrid Style Editor](#).

## Column Tasks

Clicking **Column Tasks** opens the **Column Tasks** menu. For details on the **Column Tasks** menu, see [Column Tasks Menu](#).

## About C1FlexGrid

Clicking **About C1FlexGrid** displays a dialog box, which is helpful in finding the version number of [C1FlexGrid](#).

## Dock in parent container/Undock in parent container

Clicking **Dock in parent container** sets the **Dock** property for C1FlexGrid to **Fill**.

If C1FlexGrid is docked in the parent container, the option to undock C1FlexGrid from the parent container will be available. Clicking **Undock in parent container** sets the **Dock** property for C1FlexGrid to **None**.

## Column Tasks Menu

The **Column Tasks** menu allows you to set the column caption, data field, data type, edit mask, format string, and combo list for a column, as well as set the following properties: [AllowSorting](#), [AllowEditing](#), [AllowResizing](#), [AllowDragging](#), [AllowMerging](#), and [Visible](#).

To access the **Column Tasks** menu, either click on a column in the grid or select **Column Tasks** from the **C1FlexGrid Tasks** menu.

## Column Caption

Entering a caption into the **Column Caption** box set the [Caption](#) property for the column.

## Data Field

Clicking the drop-down arrow in the **Data Field** box opens a list of available fields in the data source.

## Data Type

Clicking the drop-down arrow in the **Data Type** box opens a list of available data types.

## Edit Mask

Clicking the **ellipsis** button in the **Edit Mask** box opens the **Input Mask** dialog box.

## Format String

Clicking the **ellipsis** button in the **Format String** box opens the **Format String** dialog box.

## Combo List

Clicking the **ellipsis** button in the **Combo List** box opens the **Combo List** dialog box.

## Allow Sorting

Selecting the **Allow Sorting** check box sets the `AllowSorting` property to **True**, and allows sorting in the column. The default is checked.

## Allow Editing

Selecting the **Allow Editing** check box sets the `AllowEditing` property to **True**, and allows editing in the column. The default is checked.

## Allow Resizing

Selecting the **Allow Resizing** check box sets the `AllowResizing` property to **True**, and allows resizing the column. The default is checked.

## Allow Dragging

Selecting the **Allow Dragging** check box sets the `AllowDragging` property to **True**, and allows dragging columns in the grid. The default is checked.

## Allow Merging

Selecting the **Allow Merging** check box sets the `AllowMerging` property to **True**, and allows merging in the column. The default is unchecked.

## Allow Filtering

Selecting an option from the drop-down lets you specify the filter type along each column. The available filter options are filter by value, by condition, customized, or none.

## Visible

Selecting the **Visible** check box sets the `Visible` property to **True**, and allows the column to be visible in the grid. The default is checked.

## Caption Style

Clicking **Caption Style** opens the **Caption Style** editor for the selected column, which allows you to specify the properties for the caption text, including alignment, background, and borders.

## Column Style

Clicking **Column Style** opens the **Column Style** editor for the selected column, which allows you to specify properties for the column, including text, alignment, background, and borders.

## C1FlexGrid Tasks

Clicking **C1FlexGrid Tasks** returns you to the **C1FlexGrid Tasks** menu. For details on the **C1FlexGrid Tasks** menu, see [C1FlexGrid Tasks Menu](#).

## Dock in parent container/Undock in parent container

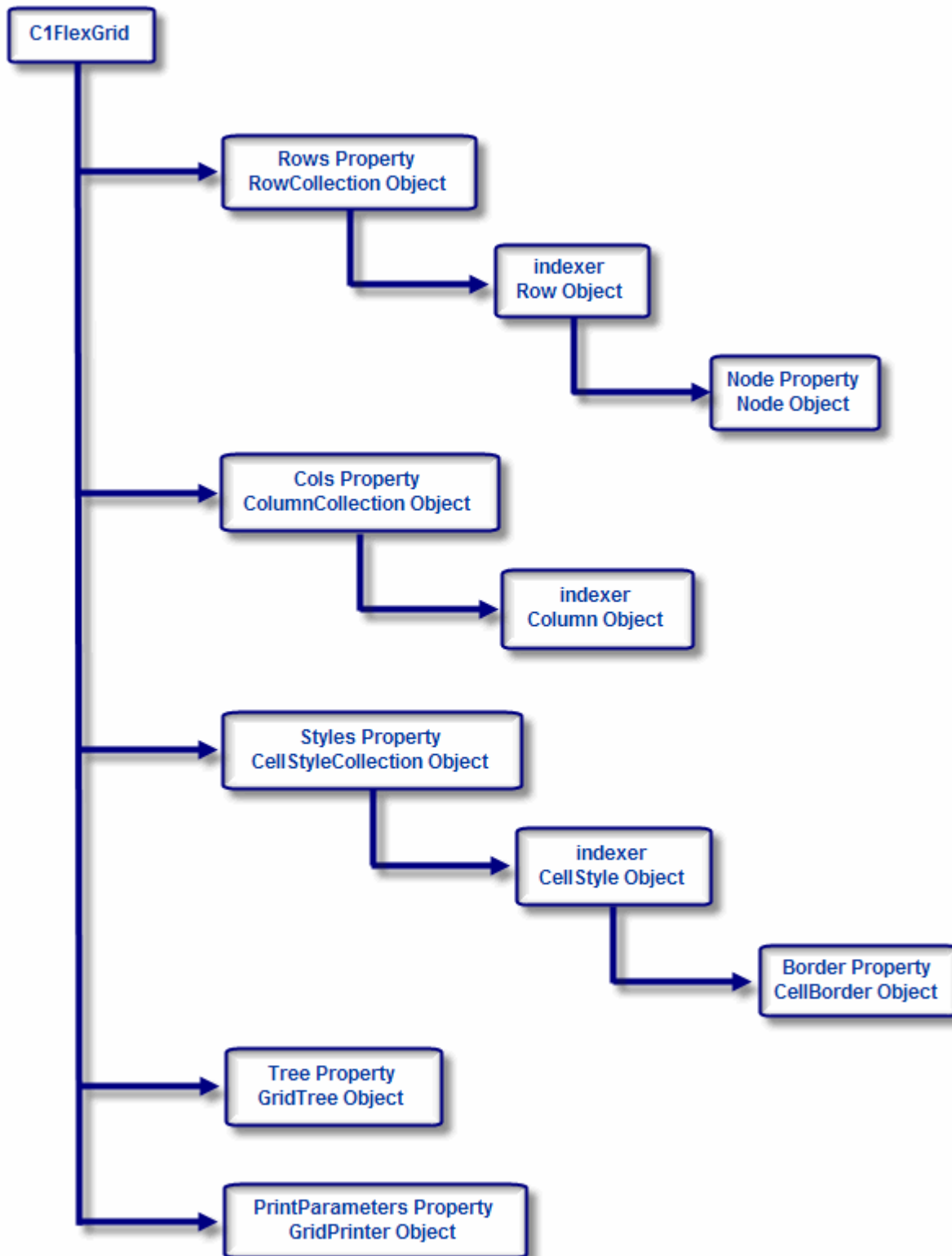
Clicking **Dock in parent container** sets the `Dock` property for `C1FlexGrid` to **Fill**.

If `C1FlexGrid` is docked in the parent container, the option to undock `C1FlexGrid` from the parent container will be available. Clicking **Undock in parent container** sets the `Dock` property for `C1FlexGrid` to **None**.

## Using the C1FlexGrid Control

The [C1FlexGrid](#) control allows you to display, edit, group and summarize data in a grid format. The grid can be bound to a data source or it can manage its own data.

The C1FlexGrid control has a rich object model with the following elements:



The following topics walk you through the main features in the C1FlexGrid control:

## [Rows and Columns](#)

Describes how to set up the grid dimensions and layout.

## [Cell Selection](#)

Describes the concepts of "current cell" and "selection".

## Cell Ranges

Describes how to work with a group of cells as a single unit.

## Cell Images

Describes how to display images in a cell.

## Formatting Cells

Describes how to customize the appearance of the grid by formatting numbers, dates, and Boolean values, or by changing fonts, colors, alignment, and pictures for individual cells or ranges.

## Editing Cells

Describes how to implement simple text editing, drop-down lists and combo lists, cell buttons, editing masks, and data validation.

## Merging Cells

Describes how to change the grid display so that cells with similar contents are merged, creating "grouped" views that highlight relationships in the data.

## Outlining and Summarizing Data

Describes how to add subtotals to grids and how to build outline trees.

## Saving, Loading, and Printing

Describes how you can save the contents or formatting of a grid and re-load it later, or exchange grid data with other applications such as Microsoft Access and Excel. This section also shows how you can print grids.

## C1FlexGrid Property Groups

Presents a map of the main C1FlexGrid properties cross-referenced by function.

## Rows and Columns

A [C1FlexGrid](#) control consists of rows and columns. The collections of rows and columns is exposed by the [Rows](#) and [Cols](#) properties.

When the grid is bound to a data source, the number of rows and columns is determined by how much data is available in the data source. In unbound mode, you can set them to arbitrary values using the **Count** property in the collections. For example, the code below sets the grid dimensions to 500 rows by 10 columns:

### To write code in Visual Basic

Visual Basic

```
_flex.Rows.Count = 500  
_flex.Cols.Count = 10
```

### To write code in C#

C#

```
_flex.Rows.Count = 500;  
_flex.Cols.Count = 10;
```

There are two basic types of rows and columns: **fixed** and **scrollable**. (The counts returned by the **Count** property include fixed and scrollable cells.) Fixed rows remain on the top of the grid when the user scrolls the grid vertically, and fixed columns remain on the left of the grid when the user scrolls the grid horizontally.

The diagram shows a table with three columns: ProductID, ProductName, and QuantityPerUnit. The first column is highlighted in orange and labeled 'Fixed Column'. The second and third columns are labeled 'Scrollable Column'. The first row is highlighted in orange and labeled 'Fixed Row'. The second and third rows are labeled 'Scrollable Row'.

ProductID	ProductName	QuantityPerUnit
1	Chai	10 boxes x 20 bags
2	Chang	24 - 12 oz bottles
3	Aniseed Syrup	12 - 550 ml bottles

Fixed cells are useful for displaying row and column header information.

The diagram shows a table with three columns: ProductID, ProductName, and QuantityPerUnit. The first column is highlighted in orange and labeled 'Caption'. The second and third columns are labeled 'Sorting Glyph'.

ProductID	ProductName	QuantityPerUnit
1	Chai	10 boxes
2	Chang	24 - 12 oz bottles
3	Aniseed Syrup	12 - 550 ml bottles
4	Chef Anton's Cajun	48 - 6 oz bottles

You can set the number of fixed rows and columns using the **Fixed** property in the Rows and Cols collections. For example, the code below creates a grid with two fixed rows and no fixed columns:

### To write code in Visual Basic

```
Visual Basic
_flex.Rows.Fixed = 1
_flex.Cols.Fixed = 0
```

### To write code in C#

```
C#
_flex.Rows.Fixed = 1;
_flex.Cols.Fixed = 0;
```

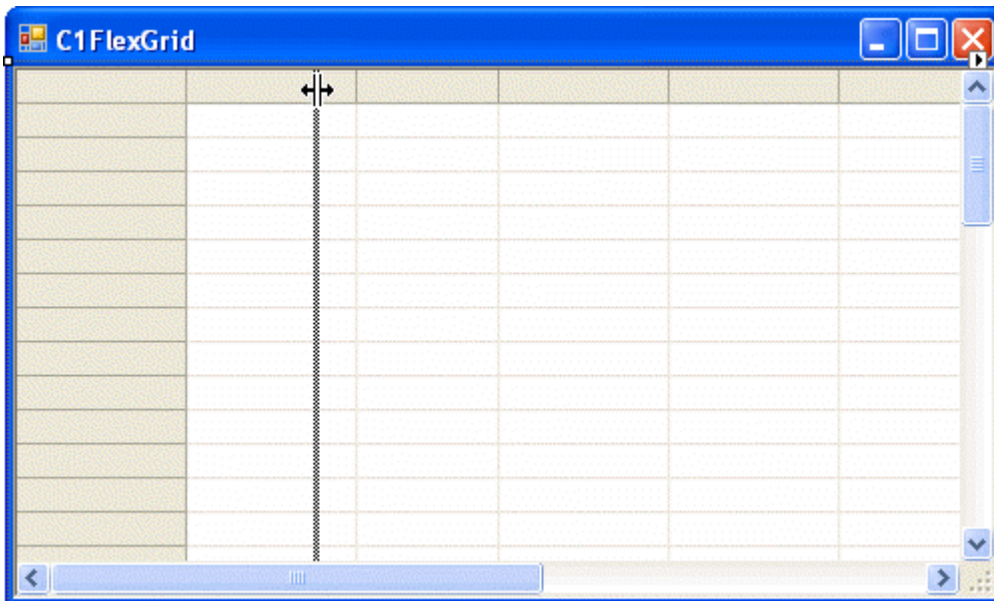
The Rows and Cols collections also contain methods for inserting, deleting, and moving rows and columns on the grid. You can use their [Item](#) property (an indexer) to access individual elements (rows and columns) in each collection.

If you prefer, you can set up the grid columns at design time instead of writing code to do it using the **C1FlexGrid Column Editor**. For details on editing columns using the **C1FlexGrid Column Editor**, see [C1FlexGrid Column Editor](#).

## Column Sizing

The width of a column is determined by its [Width](#) property. At design time, the Width property can be set directly in the grid or through the **C1FlexGrid Column Editor**. In the grid, clicking and dragging the horizontal double arrow that appears at the right edge of a column's header allows the column to be resized.





The dotted vertical line indicates how the grid will be resized. Dragging the pointer to the left makes the column smaller; dragging it to the right makes the column larger. The column's `Width` property will be adjusted when the resize operation is complete.

In the **C1FlexGrid Column Editor** or in code, specify the value of the `Width` property for a column. For details on the **C1FlexGrid Column Editor**, see [C1FlexGrid Column Editor](#). The following code sets the `Width` property of `Column1` to `10`:

#### To write code in Visual Basic

Visual Basic

```
_flex.Cols(1).Width = 10
```

#### To write code in C#

C#

```
_flex.Cols(1).Width = 10
```

To prevent resizing a specific column, set the `AllowResizing` property for the column to **False** either in the **Column Tasks** menu or **C1FlexGrid Column Editor**, or in code. For details on the **Column Tasks** menu, see [Column Tasks Menu](#). The following code sets the `AllowResizing` property for `Column1` to **False**:

#### To write code in Visual Basic

Visual Basic

```
_flex.Cols(1).AllowResizing = False
```

#### To write code in C#

C#

```
_flex.Cols(1).AllowResizing = False
```

## Star Sizing

Star-sizing is a powerful and flexible feature that allows you to specify how the total width of a grid has to be distributed among columns. It allows you to extend any set of columns and specify how the space should be distributed among them. FlexGrid allows you to specify the sizing for its columns through `StarWidth` property of the `Column` class.

For example, if you have a grid with six columns and their `StarWidth` property is set to `"*"`, `"2*"`, `"4*"`, `"*"`, `"*"`, and `"*"`, which fill the entire width of the FlexGrid control. The second column will have twice the width of the first and last three columns in the grid and the third column will be four times wider than the first and last three columns in the grid. Also, if you resize the form to make the grid wider, the columns resize automatically to keep the provided proportions.

Moreover, if you want to restrict the columns from getting too narrow or wide, FlexGrid provides `MinWidth` and `MaxWidth` properties to limit the star sizing.

The following image shows FlexGrid with the column widths set through star sizing.

ID	Tran_Type	Tran_Name	Estimated_'	Actual_Val	Tran_M ^
1	Income	Net sales	60000	54000	Income
2	Income	Interest Income	3000	3000	Income
3	Income	Asset Sales	300	450	Income
4	Personnel Expenses	Wages	9500	9600	Expense
5	Personnel Expenses	Employee benefits	4000	0	Expense
6	Personnel Expenses	Commission	5000	4500	Expense
7	Operating Expenses	Advertising	3000	2500	Expense
8	Operating Expenses	Bad debts	2000	2000	Expense
9	Operating Expenses	Cash Discounts	1500	2175	Expense
10	Operating Expenses	Delivery costs	2000	1500	Expense
11	Operating Expenses	Depreciation	1000	1000	Expense
12	Operating Expenses	Dues and subscriptions	500	525	Expense
13	Operating Expenses	Insurance	1300	1275	Expense
14	Operating Expenses	Interest	2000	2200	Expense
15	Operating Expenses	Legal and auditing	1000	800	Expense
16	Operating Expenses	Maintenance and repairs	4500	4600	Expense
17	Operating Expenses	Office supplies	800	750	Expense
18	Operating Expenses	Postage	400	350	Expense
19	Operating Expenses	Rent or mortgage	4100	4500	Expense
20	Operating Expenses	Sales expenses	350	400	Expense
21	Operating Expenses	Shipping and storage	900	840	Expense
22	Operating Expenses	Supplies	5000	4500	Expense

The following code illustrates the use of `StarWidth` property to specify star sizing for the columns.

### Visual Basic

```

c1FlexGrid1.Cols(1).StarWidth = "*"
c1FlexGrid1.Cols(2).StarWidth = "2*"
c1FlexGrid1.Cols(3).StarWidth = "4*"
c1FlexGrid1.Cols(4).StarWidth = "*"
c1FlexGrid1.Cols(5).StarWidth = "*"
c1FlexGrid1.Cols(6).StarWidth = "*"
    
```

```
'Setting the MinWidth property to prevent the column from getting too narrow
c1FlexGrid1.Cols(1).MinWidth = 80
```

- C#

```
c1FlexGrid1.Cols[1].StarWidth = "*";
c1FlexGrid1.Cols[2].StarWidth = "2*";
c1FlexGrid1.Cols[3].StarWidth = "4*";
c1FlexGrid1.Cols[4].StarWidth = "*";
c1FlexGrid1.Cols[5].StarWidth = "*";
c1FlexGrid1.Cols[6].StarWidth = "*";
```

```
//Setting the MinWidth property to prevent the column from getting too narrow
c1FlexGrid1.Cols[1].MinWidth = 80;
```

## Column Footers

Column footer is basically the bottom cell of a column which displays additional information applying to the complete column. You can display a column footer in the FlexGrid control through Footers property of the C1FlexGrid class. You can also choose to let the footer scroll along with the rows or fix it at the bottom of the grid using Fixed property of the Footer class. Furthermore, you can display information in the footer by accessing the FooterDescription collection using Descriptions property of the Footer class and adding information to it. This information can contain text with aggregates wherein text can be added to the description using Caption property of the FooterDescription class. On the other hand, to add aggregated content on data displayed in a column through aggregate functions, you first need to access the AggregateDefinition collection using Aggregates property of the FooterDescription class. Then, you can set an aggregate for a cell using Aggregate property of the AggregateDefinition class. This property accepts the value from the AggregateEnum enumeration that specifies the type of aggregate function.

The following code adds a column footer at the bottom of the FlexGrid control displaying text with aggregate content:

### Visual Basic

```
c1FlexGrid1.Footers.Descriptions.Add(new FooterDescription() { Caption = "Total" })
c1FlexGrid1.Footers.Descriptions[0].Aggregates.Add(new
AggregateDefinition()
    { Column = 4, Aggregate = AggregateEnum.Sum })
```

- C#

```
c1FlexGrid1.Footers.Descriptions.Add(new FooterDescription() { Caption = "Total" });
c1FlexGrid1.Footers.Descriptions[0].Aggregates.Add(new AggregateDefinition()
{ Column = 4, Aggregate = AggregateEnum.Sum });
```

## Cell Selection

The grid has a **cursor cell**, which displays a focus rectangle while the grid is active. The user may move the cursor with the keyboard or the mouse, and edit the contents of the cell if the grid is editable.

AtomicNumber	Element	Symbol
1	Hydrogen	H
2	Helium	He
3	Lithium	Li
4	Beryllium	Be
5	Boron	B

Cursor Cell

Notice that the Office Visual Styles also indicate the location of the cursor cell by highlighting the row and column headers of the cursor cell's position. For more information about setting the Visual Style, see [Customizing Appearance Using Visual Styles](#).

You can get or set the current cell in code using the `Row` and `Col` properties. Setting either of these properties to `-1` hides the cursor.

The grid supports **extended selections**, rectangular ranges of cells defined by two opposing corners: the cursor cell and the cell selection cell.

AtomicNumber	Element	Symbol	AtomicMass	Group
1	Hydrogen	H	1.0079400	1
2	Helium	He	4.0026020	18
3	Lithium	Li	6.9410000	1
4	Beryllium	Be	9.0121820	2
5	Boron	B	10.8110000	13
6	Carbon	C	12.0107000	14
7	Nitrogen	N	14.0067000	15

Notice that the Office Visual Styles also indicate the location of the extended selection by highlighting the row and column headers of the selected cells. For more information about setting the Visual Style, see [Customizing Appearance Using Visual Styles](#).

You can get or set the selection cell in code using the `RowSel` and `ColSel` properties, or by using the `Select` method.

**Note:** When the cursor cell changes, the selection is automatically reset. To create extended selections in code, either set `Row` and `Col` before `RowSel` and `ColSel`, or use the `Select` method.

The appearance of the selection is controlled by the following properties:

- `FocusRect` determines the type of focus rectangle that is drawn to indicate the cursor cell.
- `HighLight` determines when the selection should be highlighted (always, when the control has the focus, or never).
- `HighLight` and `Focus` are cell styles that determine the appearance of the selection (font, color, and border).

The type of selection available is determined by the `SelectionMode` property. By default, the grid supports regular range selections. You can modify this behavior to prevent extended selections, to select by row, by column, or in listbox mode (listbox mode allows you to select individual rows).

When using the listbox selection mode, you can get or set the selection status for individual rows using the `Selected` property. You can also retrieve a collection of selected rows using the `Selected` property. For example, the code below selects all rows that satisfy a condition:

### To write code in Visual Basic

```
Visual Basic
'Selects all rows with more than 8000 sales in the Sales column.
_flex.SelectionMode = C1.Win.C1FlexGrid.SelectionModeEnum.ListBox
Dim index As Integer
For index = _flex.Rows.Fixed To _flex.Rows.Count - 1
    If Val(_flex(index, "Sales")) > 80000 Then
        _flex.Rows(index).Selected = True
    End If
Next

Console.WriteLine("There are now {0} rows selected", _flex.Rows.Selected.Count)
```

### To write code in C#

```
C#
// Selects all rows with more than 8000 sales in the Sales column.
_flex.SelectionMode = SelectionModeEnum.ListBox;
for (int index = _flex.Rows.Fixed; index < _flex.Rows.Count; index++)
{
    if
    (Microsoft.VisualBasic.Conversion.Val(System.Runtime.CompilerServices.RuntimeHelpers.GetObjectValue(_flex[index,
    "Sales"]))) > 80000)
    {
        _flex.Rows[index].Selected = true;
    }
}

Console.WriteLine("There are now {0} rows selected", _flex.Rows.Selected.Count);
```

## Cell Ranges

[CellRange](#) objects allow you to work on arbitrary groups of cells as a single unit. For example, the code below creates a [CellRange](#) object, clears the data in the range, and assigns it a custom style:

### To write code in Visual Basic

#### Visual Basic

```
Dim rg As CellRange = _flex.GetCellRange(3, 3, 10, 10)
rg.Data = Nothing
rg.Style = _flex.Styles("MyRangeStyle")
```

### To write code in C#

#### C#

```
CellRange rg = _flex.GetCellRange(3, 3, 10, 10);
rg.Data = null;
rg.Style = _flex.Styles["MyRangeStyle"];
```

The [CellRange](#) object has a [StyleNew](#) property that retrieves the range style, if one exists, or creates a new one, assigns it to the range, and returns it. This property is convenient in situations where you don't need full-fledged control over formatting. For example, if all you want to do is give the range a red background, you can write:

### To write code in Visual Basic

#### Visual Basic

```
Dim rg As CellRange = _flex.GetCellRange(3, 3, 10, 10)
rg.StyleNew.BackColor = Color.Red
```

### To write code in C#

#### C#

```
CellRange rg = _flex.GetCellRange(3, 3, 10, 10);
rg.StyleNew.BackColor = Color.Red;
```

## Cell Images

Each grid cell can display images in addition to the data stored in the cell. This can be done in two ways:

- You can assign an **Image** object to a cell using the [SetCellImage](#) method. This method allows you to assign arbitrary images to each cell, and is useful if the images are not related to the data in the cell. For example, you may want to use a picture as an indicator that the data in the cell is invalid.
- You can assign an [ImageMap](#) to the column and the grid will automatically map the cell data into a corresponding image. This method is useful in situations where the image contains a representation of the data. For example, the images may contain icons that represent product types.

## Formatting Cells

One of the main strengths of the [C1FlexGrid](#) control is the ability to customize almost every aspect of the appearance of the entire grid and individual cells.

## Cell Content

To control how the content of the cells is formatted, set the [Format](#) property to a format string similar to the ones you use with the **String.Format** method in the .NET framework. For example, the code below shows short dates on column one and currency values on column two:

### To write code in Visual Basic

```
Visual Basic
' Short date.
_flex.Cols(1).Format = "d"

' Currency.
_flex.Cols(2).Format = "c"
```

### To write code in C#


```
C#
// Short date.
_flex.Cols[1].Format = "d";

// Currency.
_flex.Cols[2].Format = "c";
```

The formatting of cell content can also be set at design time using the **Format String** dialog box.

The **Format String** dialog box can be accessed through the **Column Tasks** menu or through the **C1FlexGrid Column Editor**.

- In the **Column Tasks** menu, click the **ellipsis** button in the **Format String** box.
- In the **C1FlexGrid Column Editor**, locate the Format property in the left pane, and click the **ellipsis** button next to it.

 **Note:** The **Format String** dialog box is column specific and will only change the Format property of the selected column.

You can also use custom formats like the ones used in the Visual Basic **Format** function (for example, "#,###", and so on).

## Retrieving Cell Data

You can retrieve the raw grid data using the indexers or the [GetData](#) method. To retrieve the formatted data, use the [GetDataDisplay](#) method instead. For example:

### To write code in Visual Basic

```
Visual Basic
' Short date.
_flex.Cols(1).Format = "d"

' Currency.
_flex.Cols(2).Format = "c"
```

```
_flex(1, 2) = 10000
Console.WriteLine("Raw value: {0}", _flex(1, 2))
Console.WriteLine("Display value: {0}", _flex.GetDataDisplay(1, 2))

' Raw value: 10000
' Display value: $10,000.00
```

### To write code in C#

```
C#

// Short date.
_flex.Cols[1].Format = "d";

// Currency.
_flex.Cols[2].Format = "c";

_flex[1, 2] = 10000;
Console.WriteLine("Raw value: {0}", _flex[1, 2]);
Console.WriteLine("Display value: {0}", _flex.GetDataDisplay(1, 2));

// Raw value: 10000
// Display value: $10,000.00
```

## Cell Appearance

The appearance of the cells (alignment, font, colors, borders, and so on) is handled with [CellStyle](#) objects. The grid has a [Styles](#) property that holds the collection of styles used to format the grid. This collection has some built-in members that define the appearance of grid elements, such as fixed and scrollable cells, selection, focus cell, and so on. You can change these styles to modify the way the grid looks, and you can also create your own custom styles and assign them to cells, rows, or columns.

Changing the built-in styles is the simplest way to change the appearance of the grid. For example, the code below displays the selection as bold green characters over a red background:

### To write code in Visual Basic

```
Visual Basic

Dim cs As CellStyle = _flex.Styles.Highlight
cs.Font = New Font(_flex.Font, FontStyle.Bold)
cs.ForeColor = Color.Green
cs.BackColor = Color.Red
```

### To write code in C#

```
C#

CellStyle cs = _flex.Styles.Highlight;
cs.Font = new Font(_flex.Font, FontStyle.Bold);
cs.ForeColor = Color.Green;
cs.BackColor = Color.Red;
```

You can also create your own styles and assign them to cells, rows and columns. For example, the code below creates a custom cell style and assigns it to every fifth row:

## To write code in Visual Basic

### Visual Basic

```
Dim cs As CellStyle = _flex.Styles.Add("Fifth")
cs.BackColor = Color.Gray
Dim index%
For index = _flex.Rows.Fixed To _flex.Rows.Count - 1 Step 5
    _flex.Rows(index).Style = cs
Next
```

## To write code in C#

### C#

```
CellStyle cs = _flex.Styles.Add("Fifth");
cs.BackColor = Color.Gray;
for (int index = _flex.Rows.Fixed ; index <= _flex.Rows.Count - 1; index += 5)
{
    _flex.Rows[index].Style = cs;
}
```

Here's an example that shows how you can create custom styles and assign them to columns, rows, and cell ranges:

## To write code in Visual Basic

### Visual Basic

```
' Create a new custom style
Dim s As CellStyle = _flex.Styles.Add("MyStyle")
s.BackColor = Color.Red
s.ForeColor = Color.White

' Assign the new style to a column.
_flex.Cols(3).Style = _flex.Styles("MyStyle")

' Assign the new style to a row.
_flex.Rows(3).Style = _flex.Styles("MyStyle")

' Assign the new style to a cell range.
Dim rg As CellRange = _flex.GetCellRange(4, 4, 6, 6)
rg.Style = _flex.Styles("MyStyle")
```

## To write code in C#

### C#

```
// Create a new custom style.
CellStyle s = _flex.Styles.Add("MyStyle");
s.BackColor = Color.Red;
s.ForeColor = Color.White;
```



```
// Assign the new style to a column.
_flex.Cols[3].Style = _flex.Styles["MyStyle"];

// Assign the new style to a row.
_flex.Rows[3].Style = _flex.Styles["MyStyle"];

// Assign the new style to a cell range.
CellRange rg = _flex.GetCellRange(4, 4, 6, 6);
rg.Style = _flex.Styles["MyStyle"];
```

If you prefer, you can set up styles at design time using the **C1FlexGrid Style Editor** instead of writing code to do it. For details on customizing cell appearance with the **C1FlexGrid Style Editor**, see [C1FlexGrid Style Editor](#).

## Conditional Formatting

To format cells based on their contents, you can use the [CellChanged](#) event to select a style for the cell based on its contents. For example, the code below creates a new style for large currency values and applies it to cells based on their contents:

### To write code in Visual Basic

#### Visual Basic

```
Dim cs As Cl.Win.C1FlexGrid.CellStyle

Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs)
    Handles MyBase.Load

    ' Create a custom style for large values.
    cs = _flex.Styles.Add("LargeValue")
    cs.Font = New Font(Font, FontStyle.Italic)
    cs.BackColor = Color.Gold
End Sub

' Format cells based on their content.
Private Sub _flex_CellChanged(ByVal sender As Object, ByVal e As RowColEventArgs)
    Handles _flex.CellChanged

    ' Mark currency values > 50,000 as LargeValues (reset others by setting their
    Style to Nothing).
    Dim cs As CellStyle
    If _flex(e.Row, e.Col).ToString >= 50000 Then
        cs = _flex.Styles("LargeValue")
        _flex.SetCellStyle(e.Row, e.Col, cs)
    End If
End Sub
```

### To write code in C#

#### C#

```
CellStyle cs;
```

```
private void Form1_Load(object sender, EventArgs e)
{
    // Create a custom style for large values.
    cs = _flex.Styles.Add("LargeValue");
    cs.Font = new Font(Font, FontStyle.Italic);
    cs.BackColor = Color.Gold;
}

// Format cells based on their content.
private void _flex_CellChanged( object sender, RowColEventArgs e)
{
    // Mark currency values > 50,000 as LargeValues reset others by setting their
    // Style to null).

    if (Microsoft.VisualBasic.CompilerServices.Conversions.ToDouble(_flex[e.Row,
e.Col].ToString()) >= 50000)
    {
        cs = _flex.Styles["LargeValue"];
        _flex.SetCellStyle(e.Row, e.Col, cs);
    }
}
```

## Owner-Drawn Cells

Even though the [CellStyle](#) objects offer a lot of control over the cell appearance (back and foreground colors, alignment, font, margins, borders, and so on), sometimes that is not enough. You may want to use a gradient background, or draw some custom graphics directly into a cell. In these cases, you can use the [DrawMode](#) property and the [OwnerDrawCell](#) event to gain total control over how each cell is drawn.

The [DrawMode](#) property determines whether or not the [OwnerDrawCell](#) event is fired. The event allows you to override every visual aspect of the cell. The parameters in the [OwnerDrawCell](#) event allow you to change the data that is displayed, the style used to display the data, or to take over completely and draw whatever you want into the cell.

You can change the text and image that will be shown in the cell by setting the *e.Text* and *e.Image* parameters in the event handler. You can also change the style that will be used to display the cell by setting the **e.Style** property.

Note that you should not modify the properties of the *Style* parameter because that might affect other cells. Instead, assign a new **CellStyle** object to the *Style* parameter. For example, instead of setting **e.Style.ForeColor = Color.Red**, assign a whole new style to the parameter using **e.Style = \_flex.Styles["RedStyle"]**.

You can also use your own drawing code to draw into the cell, and even combine your custom code with calls to the **e.DrawCell** method. For example, you could paint the cell background using GDI calls and then call **e.DrawCell** to display the cell borders and content.

 **Note:** For an example of advanced Owner-Draw Cells, see the **RtfGrid** sample on [GrapeCity website](#).

The code below shows an example. It uses a gradient brush to paint the background of the selected cells. First, the code sets the [DrawMode](#) property, declares a **LinearGradientBrush** object and updates the brush whenever the grid is resized:

### To write code in Visual Basic

```
Visual Basic
```

```
Dim m_GradientBrush As System.Drawing.Drawing2D.LinearGradientBrush

Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles MyBase.Load

    ' Brush to use with owner-draw cells.
    m_GradientBrush = New
System.Drawing.Drawing2D.LinearGradientBrush(ClientRectangle, Color.SteelBlue,
Color.White, 45)

    ' Use owner-draw to add gradients.
    _flex.DrawMode = C1.Win.C1FlexGrid.DrawModeEnum.OwnerDraw
End Sub

Private Sub _flex_Resize(ByVal sender As Object, ByVal e As System.EventArgs) Handles
_flex.Resize

    ' Update gradient brush when the control is resized.
    m_GradientBrush = New
System.Drawing.Drawing2D.LinearGradientBrush(ClientRectangle, Color.SteelBlue,
Color.White, 45)
End Sub
```

### To write code in C#

```
C#
System.Drawing.Drawing2D.LinearGradientBrush m_GradientBrush;

private void Form1_Load(object sender, EventArgs e)
{

    // Brush to use with owner-draw cells.
    m_GradientBrush = new
System.Drawing.Drawing2D.LinearGradientBrush(ClientRectangle, Color.SteelBlue,
Color.White, 45);

    // Use owner-draw to add gradients.
    _flex.DrawMode = DrawModeEnum.OwnerDraw;
}

private void _flex_Resize( object sender, System.EventArgs e)
{

    // Update gradient brush when the control is resized.
    m_GradientBrush = new
System.Drawing.Drawing2D.LinearGradientBrush(ClientRectangle, Color.SteelBlue,
Color.White, 45);
}
```

The second step is handling the `OwnerDrawCell` event and using the custom brush for painting the cell background. In this example, the foreground is handled by the grid itself, using the **DrawCell** method in the event argument:

**To write code in Visual Basic**

## Visual Basic

```
Private Sub _flex_OwnerDrawCell(ByVal sender As Object, ByVal e As
C1.Win.C1FlexGrid.OwnerDrawCellEventArgs) Handles _flex.OwnerDrawCell

    ' Draw the selected cell background using gradient brush.
    If _flex.Selection.Contains(e.Row, e.Col) Then

        ' Draw the background.
        e.Graphics.FillRectangle(m_GradientBrush, e.Bounds)

        ' Let the grid draw the content.
        e.DrawCell(C1.Win.C1FlexGrid.DrawCellFlags.Content)

        ' We're done drawing this cell.
        e.Handled = True
    End If
End Sub
```

**To write code in C#**

## C#

```
private void _flex_OwnerDrawCell( object sender, OwnerDrawCellEventArgs e)
{
    // Draw the selected cell background using gradient brush.
    if (_flex.Selection.Contains(e.Row, e.Col))
    {
        // Draw the background.
        e.Graphics.FillRectangle(m_GradientBrush, e.Bounds);

        // Let the grid draw the content.
        e.DrawCell(DrawCellFlags.Content);

        // We're done drawing this cell.
        e.Handled = true;
    }
}
```

## Editing Cells

By default, the [C1FlexGrid](#) control allows users to edit cells by typing into them. You can prevent users from editing the grid by setting the [AllowEditing](#) property to **False**. You can also prevent users from editing specific columns by settings the [AllowEditing](#) property to **False**. (When the grid is bound to a data source, it detects which columns are editable and automatically sets the [AllowEditing](#) property.)

To start editing a cell, the user can:

- Start typing into the cell. This replaces the contents of the cell.
- Press F2 or Enter. This puts the grid in edit mode and puts the current cell contents in the editor.
- Double-click a cell. This has the same effect as pressing F2, but the cursor appears where the cell is clicked.

The basic editing mode allows users to type values into the cells. If the column being edited has a specific data type, values entered by the user are converted into the proper data type automatically. If the user types a value that cannot be converted into the proper data type, the grid fires a [GridError](#) event and ignores the edits.

The basic editing is sufficient for many applications, but the C1FlexGrid has properties and events that allow you to control the editing process and provide selection lists, editing buttons, and advanced validation control.

Starting with version 2.5, the C1FlexGrid also has built-in support for external editors. This allows you to use any control as a grid editor (for example, you can now use the **C1Input** controls as grid editors).

These features are described in the following topics.

## Lists and Combos

In many applications, cells have a well-defined list of possible values. In these cases, you can let users select the value from a **drop-down list**. To do this, build a string containing all the choices separated by pipe characters (for example, "True|False|Don't know") and assign it to the [ComboList](#) property. Each column may have a different list. Setting the ComboList property causes the grid to display a drop-down box next to the cell. The user can click the box (or press F2) to display the list of choices available for that cell.

Another common situation is where cells have a list of common values, but users should be allowed to type something else as well. This can be accomplished with **drop-down combos**, a combination of text box and drop-down list. To create combos, just start the choice list with a pipe character (for example "|True|False|Don't know"), then assign it to the ComboList property as before.

For example, the code below would cause the grid to display a drop-down combolist containing color names on column one, and a drop-down combo on column two. When editing column one, the user must pick a value from the list. When editing column two, the user can pick a value or type in something else:

### To write code in Visual Basic

#### Visual Basic

```
' Drop-down list.
_flex.Cols(1).ComboList = "Red|Green|Blue|Red|White"

' Drop-down combo.
_flex.Cols(2).ComboList = "|Red|Green|Blue|Red|White"
```

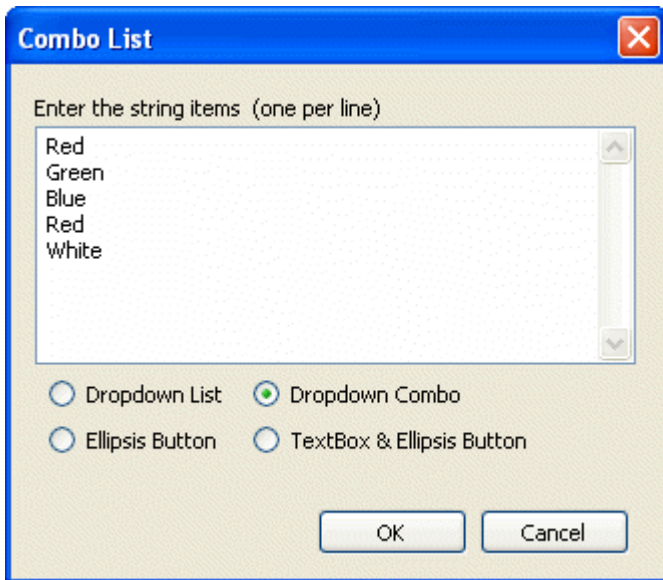
### To write code in C#

#### C#

```
// Drop-down list.
_flex.Cols[1].ComboList = "Red|Green|Blue|Red|White";


// Drop-down combo.
_flex.Cols[2].ComboList = "|Red|Green|Blue|Red|White";
```

The ComboList property can also be set at design time using the **Combo List** dialog box. The **Combo List** dialog box allows you to choose if you want the list to appear as a **Dropdown List**, **Dropdown Combo**, **Ellipsis Button**, or **TextBox and Ellipsis Button**.



The **Combo List** dialog box can be accessed through the **Column Tasks** menu or through the **C1FlexGrid Column Editor**.

- In the **Column Tasks** menu, click the **ellipsis** button in the **Combo List** box.
- In the **C1FlexGrid Column Editor**, locate the **ComboList** property in the left pane, and click the **ellipsis** button next to it.

 **Note:** The **Combo List** dialog box is column specific and will only change the **ComboList** property of the selected column.

In some cases, cells in the same column may need different lists. For example, a property list may show properties on the first column and their values on the second. The values depend on the property, so valid choices change from one row to the next. In these cases, you should trap the [BeforeEdit](#) event and set the **ComboList** property to the appropriate list for the current cell. The **ComboList** property applies to the whole grid.

The built-in **ComboBox** provides an auto-search feature by default. As the user types a value, the selection will move to the next match. You can disable this feature using the [EditOptions](#) property and control the time before the grid resets the auto-search buffer using the [AutoSearchDelay](#) property.

The built-in **ComboBox** also has an auto-cycle feature like the editors in the Visual Studio Properties window. When you double-click a cell that has a list associated with it, the grid will automatically select the next value. You can also disable this feature using the [EditOptions](#) property.

## Checkboxes

By default, the grid displays values in Boolean columns as checkboxes (the type of the column is determined by the [DataType](#) property of the [Column](#) object). If you don't want Boolean values displayed as checkboxes, set the column's [Format](#) property to a string containing the values that should be displayed for True and False values. For example:

### To write code in Visual Basic

```
Visual Basic
_flex.Cols["bools"].Format = "Yes;No"
```

### To write code in C#

C#

```
_flex.Cols["bools"].Format = "Yes;No";
```

In unbound mode, you can use the `GetCellCheck` and `SetCellCheck` properties to add checkboxes to any cells. The checkboxes will be displayed along with any text in the cell, and you can set their position using the column's `ImageAlign` property.

There are two types of check boxes: Boolean and tri-state. Boolean check boxes toggle between the **CheckEnum.Checked** and **CheckEnum.Unchecked** states. Tri-state check boxes cycle through the settings **CheckEnum.TSChecked** and **CheckEnum.TSUnchecked** and **CheckEnum.TSGrayed**.

If the cell has a check box and the `AllowEditing` property is set to **True**, the user can change the state of the check boxes by clicking them with the mouse or by pressing the SPACE bar or ENTER keys.

By default, toggling the value of checkbox with the mouse or keyboard will toggle all selected checkboxes. You can disable this feature using the `EditOptions` property.

## Value-Mapped Lists

The `ComboBox` property described above ensures that cell values are selected from a list. The value selected by the user is converted into the appropriate type for the column and stored in the grid, exactly as if the user had typed the value.

In many cases, cells can assume values from well-defined lists, but you want to display a user-friendly version of the actual value. For example, if a column contains product codes, you may want to store the code but display the product name instead.

This is accomplished with the `DataMap` property. This property contains a reference to an **IDictionary** object that establishes the mapping between what is stored in the grid and what is visible to the user (the `IDictionary` interface is defined in the `System.Collections` namespace, and is implemented by the `Hashtable` class among others).

For example, the code below creates a data map that contains color values and their names. The colors are stored in the grid, and their names are displayed to the user:

### To write code in Visual Basic

Visual Basic

```
Dim dtMap As Hashtable = New Hashtable()  
dtMap.Add(Color.Red, "Apple")  
dtMap.Add(Color.Green, "Forest")  
dtMap.Add(Color.Blue, "Sky")  
dtMap.Add(Color.Black, "Coal")  
dtMap.Add(Color.White, "Snow")  
_flex.Cols(1).DataType = GetType(Color)  
_flex.Cols(1).DataMap = dtMap
```

### To write code in C#

C#

```
System.Collections.Hashtable dtMap = new System.Collections.Hashtable();  
dtMap.Add(Color.Red, "Apple");  
dtMap.Add(Color.Green, "Forest");  
dtMap.Add(Color.Blue, "Sky");  
dtMap.Add(Color.Black, "Coal");  
dtMap.Add(Color.White, "Snow");
```

```
_flex.Cols[1].DataType = typeof(Color);  
_flex.Cols[1].DataMap = dtMap;
```

Any class that implements **IDictionary** can be used as a **DataMap**. For example, **Hashtable**, **ListDictionary**, and **SortedList** all provide valid data maps. The difference is that when they are used in editable columns, the order of the items in the drop down list will depend on the class.

The **SortedList** class sorts items by key, **Hashtable** uses an arbitrary order, and **ListDictionary** keeps the order in which items were added to the list. Because of this, **ListDictionary** is usually the best choice for DataMaps.

Note that the keys in the data map must be of the same type as the cells being edited. For example, if a column contains short integers (Int16), then any data maps associated with the column should have short integer keys. Using regular integers (Int32) as keys would not work.

The example below shows the difference:

### To write code in Visual Basic

#### Visual Basic

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs)  
    Handles MyBase.Load  
  
        ' Sorts by key.  
        Dim sl As New SortedList()  
        sl.Add("0", "Zero")  
        sl.Add("1", "One")  
        sl.Add("2", "Two")  
        sl.Add("3", "Three")  
  
        ' Keeps Add order.  
        Dim ld As New Specialized.ListDictionary()  
        ld.Add(0, "Zero")  
        ld.Add(1, "One")  
        ld.Add(2, "Two")  
        ld.Add(3, "Three")  
  
        ' Arbitrary order.  
        Dim ht As New Hashtable()  
        ht.Add(0, "Zero")  
        ht.Add(1, "One")  
        ht.Add(2, "Two")  
        ht.Add(3, "Three")  
  
        _flex.Cols(1).DataMap = sl  
        _flex.Cols(1).Caption = "SortedList"  
        _flex.Cols(2).DataMap = ld  
        _flex.Cols(2).Caption = "ListDictionary"  
        _flex.Cols(3).DataMap = ht  
        _flex.Cols(3).Caption = "HashTable"  
End Sub
```

### To write code in C#

#### C#



```
private void Form1_Load(object sender, System.EventArgs e);
{
    // Sorts by key.
    System.Collections.SortedList sl = new System.Collections.SortedList();
    sl.Add("0", "Zero");
    sl.Add("1", "One");
    sl.Add("2", "Two");
    sl.Add("3", "Three");

    // Keeps Add order.
    System.Collections.Specialized.ListDictionary ld = new
System.Collections.Specialized.ListDictionary();
    ld.Add(0, "Zero");
    ld.Add(1, "One");
    ld.Add(2, "Two");
    ld.Add(3, "Three");

    // Arbitrary order.
    System.Collections.Hashtable ht = new System.Collections.Hashtable();
    ht.Add(0, "Zero");
    ht.Add(1, "One");
    ht.Add(2, "Two");
    ht.Add(3, "Three");

    _flex.Cols[1].DataMap = sl;
    _flex.Cols[1].Caption = "SortedList";
    _flex.Cols[2].DataMap = ld;
    _flex.Cols[2].Caption = "ListDictionary";
    _flex.Cols[3].DataMap = ht;
    _flex.Cols[3].Caption = "HashTable";
}
```

Also, if the column's [DataType](#) property is set to an enumeration, the grid will automatically build and use a data map with the names of each value in the enumeration. For example, the following code creates an enumeration containing countries. The country values are stored in the grid, and their names are displayed to the user:

#### To write code in Visual Basic

##### Visual Basic

```
Private Enum Countries
    NewYork
    Chicago
    NewOrleans
    London
    Paris
End Enum

Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles MyBase.Load
    _flex.Cols(1).DataType = GetType(Countries)
End Sub
```

**To write code in C#**

```
C#  
  
private enum Countries  
{  
    NewYork,  
    Chicago,  
    NewOrleans,  
    London,  
    Paris  
}  
  
private void Form1_Load(object sender, EventArgs e)  
{  
    _flex.Cols[1].DataType = typeof(Countries);  
}
```

## Cell Buttons

Certain types of cells may require sophisticated editors other than text boxes or drop-down lists. For example, if a column contains file names or a color, it should be edited with a dialog box.

In these cases, you should set the [ComboList](#) property to an ellipsis ("..."). The control will display a button next to the cell and will fire the [CellButtonClick](#) event when the user clicks on it. You can trap the event, show the dialog box, and update the cell's contents with the user's selection.

If you add a pipe character before the ellipsis, then the user will also be allowed to edit the cell contents by typing into the cell.

By default, the cell buttons display the ellipsis. You can assign pictures to the cell buttons using the [CellButtonImage](#) property.

The example below shows how you can use cell buttons to display a color picker dialog box for choosing a color in a column.

**To write code in Visual Basic**

```
Visual Basic  
  
' Set up color column.  
Dim c As Cl.Win.ClFlexGrid.Column = _flex.Cols(1)  
c.DataType = GetType(Color)  
  
' Show cell button.  
c.ComboList = "..."
```

**To write code in C#**

```
C#  
  
// Set up color column.  
Column c = _flex.Cols[1];  
c.DataType = typeof(Color);  
  
// Show cell button.
```

```
c.ComboList = "...";
```

This code sets up the column so the user can click a button and select a color from a dialog box. The next step is the code that handles clicks on the cell button:

## To write code in Visual Basic

### Visual Basic

```
Private Sub _flex_CellButtonClick(ByVal sender As Object, ByVal e As
Cl.Win.C1FlexGrid.RowColEventArgs) Handles _flex.CellButtonClick

    ' Create color picker dialog.
    Dim clrDlg As New ColorDialog()

    ' Initialize the dialog.
    If _flex(e.Row, e.Col) Is GetType(Color) Then
        clrDlg.Color = _flex(e.Row, e.Col)
    End If

    ' Get new color from dialog and assign it to the cell.
    If clrDlg.ShowDialog() = Windows.Forms.DialogResult.OK Then
        _flex(e.Row, e.Col) = clrDlg.Color
    End If
End Sub
```

## To write code in C#

### C#

```
private void _flex_CellButtonClick( object sender, RowColEventArgs e)
{
    // Create color picker dialog.
    ColorDialog clrDlg = new ColorDialog();

    // Initialize the dialog.
    if (_flex[e.Row, e.Col] == typeof(Color))
    {
        clrDlg.Color = (Color)_flex[e.Row, e.Col];
    }

    // Get new color from dialog and assign it to the cell.
    if (clrDlg.ShowDialog() == DialogResult.OK)
    {
        _flex[e.Row, e.Col] = clrDlg.Color;
    }
}
```

## Masks

The [C1FlexGrid](#) control also supports masked editing. This type of editing uses an input mask to provide a template and automatically validate the input as the user types. The mask is specified by the [EditMask](#) property, which can be

used with regular text fields and with drop-down combo fields.

Mask strings have two types of characters: literal characters, which become part of the input, and template characters, which serve as placeholders for characters belonging to specific categories (for example, digits or alphabetic). For example, the code below assigns a "(999) 999-9999" editing mask to column one, which holds phone numbers (the digit "9" is a placeholder that stands for any digit):

### To write code in Visual Basic

#### Visual Basic

```
' Set up a phone number edit mask.
_flex.Cols(1).EditMask = "(999) 999-9999"
```

### To write code in C#

#### C#

```
// Set up a phone number edit mask.
_flex.Cols[1].EditMask = "(999) 999-9999";
```

Setting the EditMask property to a non-empty string causes it to use the built-in masked editor, even if the column contains date/time values (usually, a **DateTimePicker** control is used to edit these columns). This is especially convenient if you have **DateTime** columns that hold times only (not dates). In these cases, you can set the following properties to use a masked editor instead of the **DateTimePicker** control:

### To write code in Visual Basic

#### Visual Basic

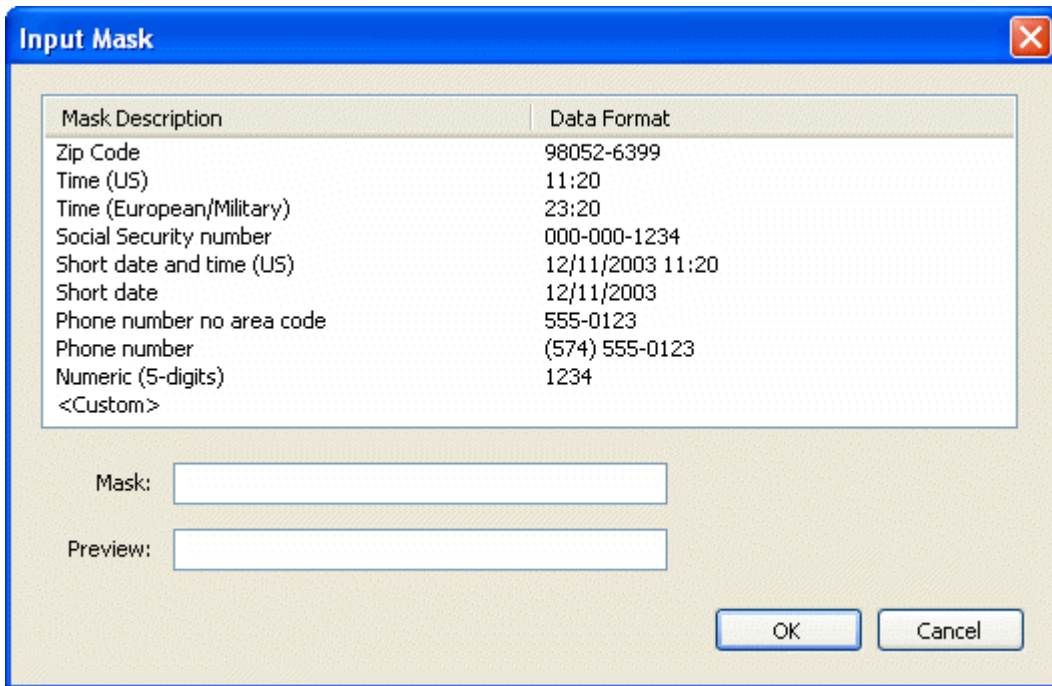
```
_flex.Cols(1).DataType = GetType(DateTime)
_flex.Cols(1).Format = "hh:mm tt"
_flex.Cols(1).EditMask = "99:99 LL"
```

### To write code in C#

#### C#

```
_flex.Cols[1].DataType = typeof(DateTime);
_flex.Cols[1].Format = "hh:mm tt";
_flex.Cols[1].EditMask = "99:99 LL";
```

The EditMask property can also be set at design time using the **Input Mask** dialog box.



The **Input Mask** dialog box can be accessed through the **Column Tasks** menu or through the **C1FlexGrid Column Editor**.

- In the **Column Tasks** menu, click the **ellipsis** button in the **Edit Mask** box.
- In the **C1FlexGrid Column Editor**, locate the EditMask property in the left pane, and click the **ellipsis** button next to it.

**Note:** The **Input Mask** dialog box is column specific and will only change the EditMask property of the selected column.

For details on the syntax used to build the mask strings, see the EditMask property in the control reference section. If different cells in the same column need different masks, trap the **BeforeEdit** event and set the EditMask property to an appropriate value for the current cell.

## Validation

In many cases, edit masks alone are not enough to ensure that the data enters by the user was valid. For example, a mask won't let you specify a range of possible values, or validate the current cell based on the contents of another cell.

In these cases, trap the **ValidateEdit** event and see if the value contained in the **Editor.Text** property is a valid entry for the current cell (at this point, the cell still has the original value in it). If the input is invalid, set the **Cancel** parameter to **True** and the grid will remain in edit mode until the user types a valid entry.

For example, the code below validates input into a currency column to ensure that values entered are between 1,000 and 10,000:

### To write code in Visual Basic

```
Visual Basic
Private Sub _flex_ValidateEdit(ByVal sender As Object, ByVal e As
C1.Win.C1FlexGrid.ValidateEditEventArgs) Handles _flex.ValidateEdit
```

```
' Validate amounts.
If _flex.Cols(e.Col).DataType Is GetType(Decimal) Then
    Try
        Dim dec As Decimal = Decimal.Parse(_flex.Editor.Text())
        If (dec < 1000) Or (dec > 10000) Then
            MessageBox.Show("Value must be between 1,000 and 10,000")
            e.Cancel = True
        End If
    Catch
        MessageBox.Show("Value not recognized as a Currency")
        e.Cancel = True
    End Try
End If
End Sub
```

### To write code in C#

```
C#
private void _flex_ValidateEdit( object sender, ValidateEditEventArgs e)
{
    // Validate amounts.
    if (_flex.Cols[e.Col].DataType == typeof(Decimal))
    {
        try
        {
            Decimal dec = Decimal.Parse(_flex.Editor.Text);
            if ( dec < 1000 || dec > 10000 )
            {
                MessageBox.Show("Value must be between 1,000 and 10,000");
                e.Cancel = true;
            }
        }
        catch
        {
            MessageBox.Show("Value not recognized as a Currency");
            e.Cancel = true;
        }
    }
}
}
```

## Custom Editors

The built-in editors provide a lot of flexibility and power, but in some cases you may want to use external controls as specialized editors. For example, you may want to use the **C1NumericInput** control that provides a drop-down calculator for entering numbers, or an editor for selecting from multi-column lists, or a specialized control that you wrote to edit your business objects.

Any control that derives from the base **Control** class can be used as a basic grid editor. Controls that implement the **IC1EmbeddedEditor** interface (defined in C1.Common.dll) can provide better integration with the grid and more

advanced features. For details on the **IC1EmbeddedEditor** interface, see the [Editor](#) property.

To use a control as a custom editor, all you have to do is associate an instance of the control with a grid column or a style using its **Editor** property. You can do this in the designer (using the **Column Editor**) or in code. After that, the control will be automatically used by the grid.

To define custom editors at design time, add an instance of the editor control to the form, then select **Designer** from the **C1FlexGrid Tasks** menu to open the **C1FlexGrid Column Editor**. Select the columns that should use the custom editor and set their Editor properties to the name of the new editor control.

For example, to use a **NumericUpDown** control as a grid editor, follow these steps:

1. Add a [C1FlexGrid](#) control to the form.
2. Add a **NumericUpDown** control to the form and set its **BorderStyle** property to **None**.
3. Select **Designer** from the **C1FlexGrid Tasks** menu. For more information on accessing the **C1FlexGrid Column Editor**, see [Accessing the C1FlexGrid Column Editor](#).
4. In the **C1FlexGrid Column Editor**, select the first scrollable grid column and set its **Editor** property to **NumericUpDown1**.

Run the project and edit some values in the first column. Notice how the grid positions and initializes the **NumericUpDown** control so you can edit cell values. When you are done editing a cell, click a different cell or press the TAB key to move to the next one. Notice how the new value is applied to the cell.

You can also assign custom editors to the grid using code:

#### To write code in Visual Basic

##### Visual Basic

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs)
    Handles MyBase.Load

    ' Create the custom editor.
    Dim editor as New NumericUpDown()
    editor.BorderStyle = BorderStyle.None

    ' Assign the custom editor to the grid.
    _flex.Cols(1).Editor = editor
End Sub
```

#### To write code in C#

##### C#

```
private void Form1_Load(object sender, System.EventArgs e)
{
    // Create the custom editor.
    NumericUpDown editor = new NumericUpDown();
    editor.BorderStyle = BorderStyle.None;

    // Assign the custom editor to the grid.
    _flex.Cols[1].Editor = editor;
}
```

## Creating Custom Editors

Any control that derives from the **Control** base class can be used as a grid editor. This is possible because the grid knows enough about the base class to access properties such as **Text** and **Bounds**, and events such as **Leave** and **TextChanged**. In many cases this level of support is adequate.

In some cases, however, you may want to use controls that do not follow the base class that closely. For example, a **DateTimePicker** control has a **Value** property that should be used to retrieve the edited value instead of **Text**. In these cases, you can implement one or more methods in the **IC1EmbeddedEditor** interface to override the default behavior. For example, all controls in the C1Input library support **IC1EmbeddedEditor** and therefore integrate closely with **C1FlexGrid** (and also **C1TrueDBGrid**).

The **IC1EmbeddedEditor** interface is fairly simple, and because the grid binds to it using late binding, you don't even have to implement all its members. Only implement the ones that make sense to your editor control.

The interface does provide enough flexibility to allow virtually any control to be used as a grid editor. You can even use **UITypeEditor** classes as grid editors. To do this, you need a wrapper class that:

1. Derives from **Control** (**UITypeEditor** doesn't).
2. Implements the **IC1EmbeddedEditor** interface.
3. Encapsulates the appropriate **UITypeEditor**.

We provide source code for this wrapper class in the **CustomEditors** sample.

Using the **UITypeEditor** wrapper class, you can use any **UITypeEditors** with the **C1FlexGrid**. .NET provides several **UITypeEditors** for editing colors, fonts, file names, and so on. You can also write your own **UITypeEditors**, in some cases that is easier than writing a control.



**Note:** For an example of using built-in, custom, and **UITypeEditor** editors with **C1FlexGrid**, see the **CustomEditors** sample on [GrapeCity website](#).

## Edit Mode

You can determine whether the grid is in edit mode by reading the value of the **Editor** property. If this property returns null, the grid is not in edit mode. Otherwise, the grid is in edit mode and the property returns a reference to the control that is being used to edit the cell (the control may be a **TextBox**, a **ComboBox**, or other type of control).

You can put the grid in edit mode programmatically using the **StartEditing** method, and finish editing using the **FinishEditing** method.

You can also set **PreserveEditMode** to retain edit mode of cells in a grid on navigation, using arrow or tab keys.

You can control the editing process further by handling the editing events fired by the grid. In the process of editing a cell, the grid fires the following sequence of events:

Event	Description
<a href="#">BeforeEdit</a>	This event fires whenever an editable cell is selected. It allows you to prevent the cell from being edited by setting the event's <i>Cancel</i> parameter to <b>True</b> . You can also modify the <b>ComboList</b> property so the appropriate drop-down button gets painted in the cell. Note that the user might not actually start editing after this, he could simply move the selection to a different cell or control.
<a href="#">StartEdit</a>	This event is similar to <b>BeforeEdit</b> , except the user has actually typed a key or clicked the dropdown button in the cell and really is about to start editing. You can still cancel the editing at this point. Note that the <b>Editor</b> property is still null at this point, because the control hasn't yet determined the type of editor that should be used. You can assign custom editors to the <b>Editor</b> property at this point.



Event	Description
<a href="#">SetupEditor</a>	This event fires after the editor control has been created and configured to edit the cell, but before it is displayed. You can change the editor properties at this point (for example, set the maximum length or password character to be used in a TextBox editor). You can also attach your own event handlers to the editor.
<a href="#">ValidateEdit</a>	This event fires when the user is done editing, before the editor value gets copied back into the grid. You can examine the original value by retrieving it from the grid (the event provides the coordinates of the cell). You can examine the new value about to be assigned to the grid through the Editor properties (for example, <b>Editor.Text</b> ). If the new value is not valid for the cell, set the <i>Cancel</i> parameter to <b>True</b> and the grid will remain in edit mode. If instead of keeping the cell in edit mode you would rather restore the original value and leave edit mode, set <b>Cancel</b> to <b>True</b> and then call the <i>FinishEditing</i> method.
<a href="#">AfterEdit</a>	This event fires after the new value has been applied to the cell and the editor has been deactivated. You can use this event to update anything that depends on the cell value (for example, subtotals or sorting).

## Grouping

Grouping refers to combining rows based on column values. When grouping is applied to a grid, it automatically sorts the data, splits it into groups, and adds collapsible group rows above or below each group. In FlexGrid, these group rows are added above each group, by default. In case, you want to add them below every group, you can use [SubtotalPosition](#) property of **C1FlexGrid** class, which accepts values from [SubtotalPositionEnum](#) enumeration.

As grid is grouped on the basis of column values, it displays grouped columns by default. However, you can remove those columns from view by setting [HideGroupedColumns](#) property of **C1FlexGrid** class to true. Also, by default the group row displays a string of "{name}:{value}" format, where name refers to the grouped column and value is one of the unique values in the column. However, this can be altered using [GroupHeaderFormat](#) property of **C1FlexGrid** class, which sets format string for the groups. The group header string is displayed in the first column of the group row. In case there is large content in the group header, it might get clipped because of the column width. This can be resolved using [AllowMerging](#) property of **C1FlexGrid** class. You can set the [AllowMerging](#) property to [AllowMergingEnum.Nodes](#) so that the group header content can spill into the adjacent empty cells.

FlexGrid also allows you to aggregate the data and display the aggregated values on group rows for one or more columns. It provides [Aggregate](#) property of the [Column](#) class that can be set on columns to show their aggregate values (like sum or average) on the group header rows.

### Sort grouped columns

FlexGrid allows you to sort columns by clicking their headers with the mouse through [AllowSorting](#) property of **C1FlexGrid** class. Sorting grouped columns will sort the groups and alter the order in which the groups are displayed. The sorting will be based on the value used to create the group. Sorting ungrouped columns will sort values within each group only. For example, if the user clicked the "Product Name" column header on the grid, the groups remain unchanged but the rows are sorted based on the alphabetical sorting applied to the products.

### Customize the appearance of outline tree

FlexGrid lets you customize the appearance of the outline tree that shows the grouping in FlexGrid by setting the [Tree.Style](#) property of FlexGrid. For example, set it to [TreeStyleFlags.Leaf](#) to show the group header text.

All these features can be combined to perform grouping in a better way. In FlexGrid, grouping can be performed using any of the following two ways:

- **Dynamic grouping:** Allows grouping without UI interaction, i.e., through code.
- **FlexGridGroupPanel control:** Allows you to perform grouping through UI interaction without using any code.

The following image shows the FlexGrid control with grouped data based on its column values.

1	* OrderID	EmployeeID	OrderDate	RequiredDate	ShippedDate	ShipVia	^
CustomerID:ALFKI							
	10643	6	25-09-2015	23-10-2015	03-10-2015		1
	10692	4	03-11-2015	01-12-2015	13-11-2015		2
	10702	4	13-11-2015	25-12-2015	21-11-2015		1
	10835	1	15-02-2016	14-03-2016	21-02-2016		3
	10952	1	15-04-2016	27-05-2016	23-04-2016		1
	11011	3	09-05-2016	06-06-2016	13-05-2016		1
CustomerID:ANATR							
	10308	7	19-10-2014	16-11-2014	25-10-2014		3
	10625	3	08-09-2015	06-10-2015	14-09-2015		1
	10759	3	29-12-2015	26-01-2016	12-01-2016		3
	10926	4	03-04-2016	01-05-2016	10-04-2016		3
CustomerID:ANTON							
	10365	3	28-12-2014	25-01-2015	02-01-2015		2
	10507	7	16-05-2015	13-06-2015	23-05-2015		1
	10535	4	13-06-2015	11-07-2015	21-06-2015		1
	10573	7	20-07-2015	17-08-2015	21-07-2015		3
	10677	1	23-10-2015	20-11-2015	27-10-2015		3
	10682	3	26-10-2015	23-11-2015	01-11-2015		2
	10856	3	28-02-2016	27-03-2016	12-03-2016		2
CustomerID:AROUT							
	10355	6	16-12-2014	13-01-2015	21-12-2014		1

The following code shows grouping in FlexGrid:

## Visual Basic

```
'Add group
flex.GroupDescriptions = new GroupDescription[] {new
GroupDescription("CustomerID")}

'Showing aggregate(sum) on the group header rows
Dim col = flex.Cols("Freight")
col.Aggregate = AggregateEnum.Sum
col.Format = "N2"

'Setting grid's AllowMerging property to Nodes so that the group header
' content can spill into adjacent empty cells
flex.AllowMerging = AllowMergingEnum.Nodes

'Setting HideGroupedColumns property to true in order to hide the
grouped columns
flex.HideGroupedColumns = True
```

```
'Customizing the string which is displayed on the group headers
flex.GroupHeaderFormat = "{name}:{value}"

'Customizing the appearance of the outline tree
flex.Tree.Style = TreeStyleFlags.CompleteLeaf
```

C#

```
//Add group
flex.GroupDescriptions = new GroupDescription[] {new
GroupDescription("CustomerID")};

//Showing aggregate(sum) on the group header rows
var col = flex.Cols["Freight"];
col.Aggregate = AggregateEnum.Sum;
col.Format = "N2";

//Setting grid's AllowMerging property to Nodes so that the group header
content can //spill into adjacent empty cells
flex.AllowMerging = AllowMergingEnum.Nodes;

//Setting HideGroupedColumns property to true in order to hide the
grouped columns
flex.HideGroupedColumns = true;

//Customizing the string which is displayed on the group headers
flex.GroupHeaderFormat = "{name}:{value}";

//Customizing the appearance of the outline tree
flex.Tree.Style = TreeStyleFlags.CompleteLeaf;
```

## Grouping through Code

FlexGrid supports grouping through code. It provides [GroupDescriptions](#) property to describe how data source items are grouped in the grid. This property accepts the instance of any collection which implements `ICollection<GroupDescription>` interface (e.g. `List<GroupDescription>`) as its value. The items of the collection describe grouping using a property name as the criterion.

The following code shows grouping in FlexGrid:

Visual Basic

```
'Add group
flex.GroupDescriptions = new GroupDescription[] {new
GroupDescription("CustomerID")}
```

C#

```
//Add group
flex.GroupDescriptions = new GroupDescription[] {new
GroupDescription("CustomerID")};
```

## Grouping through GroupPanel

FlexGridGroupPanel control is an extension to FlexGrid control which is used to create groups in FlexGrid at runtime. To show grouping in FlexGrid at runtime, bind the grid to be grouped with the FlexGridGroupPanel control using the `FlexGrid` property of the `C1FlexGridGroupPanel` class.

On adding the FlexGridGroupPanel control to the form, it displays a message "Drag a column here to group by that column". However, you can add custom message in this area by setting `Text` property of the `C1FlexGridGroupPanel` class. The class also provides the `AutoSize` property which when set to true, resizes the group panel automatically according to the message content.

At runtime, user needs to drag a column to the FlexGridGroupPanel to perform grouping in FlexGrid based on that column. This creates a group element inside FlexGridGroupPanel representing dragged column. The group element contains the column's caption and a clear button. You can hide the clear button by setting `ShowClearButton` property of the `C1FlexGridGroupPanel` class to false.

When you want to group the data by multiple columns, order of the dragged columns determines the order in which grouping will be done. Once grouping is performed, group headers get created for each group. You can also set the maximum number of groups allowed within FlexGrid using `MaxGroups` property of `C1FlexGridGroupPanel` class. By default, all the created groups appear in the expanded state. To change this setting, you can set the `AutoExpandAllGroups` of `C1FlexGridGroupPanel` class to `false`.

The following code illustrates the properties of FlexGridGroupPanel:

### Visual Basic

```
Dim flexGroupPanel As C1FlexGridGroupPanel = New C1FlexGridGroupPanel()
flexGroupPanel.Bounds = New System.Drawing.Rectangle(100, 10, 500, 130)
this.Controls.Add(flexGroupPanel)
'Setting C1FlexGridGroupPanel's general properties
flexGroupPanel.FlexGrid = _flex
flexGroupPanel.Text = "Drag the columns here.."
flexGroupPanel.MaxGroups = 5
flexGroupPanel.AutoSize = False
flexGroupPanel.AutoExpandAllGroups = True
```

### • C#

```
C1FlexGridGroupPanel flexGroupPanel = new C1FlexGridGroupPanel();
flexGroupPanel.Bounds = new System.Drawing.Rectangle(100, 10, 500, 130);
this.Controls.Add(flexGroupPanel);
//Setting C1FlexGridGroupPanel's general properties
flexGroupPanel.FlexGrid = _flex;
flexGroupPanel.Text = "Drag the columns here..";
flexGroupPanel.MaxGroups = 5;
flexGroupPanel.AutoSize = false;
flexGroupPanel.AutoExpandAllGroups = true;
```

## Customize FlexGridGroupPanel ContextMenuStrip

FlexGridGroupPanel have embedded context menu with the following options:

- Full Expand: Expands all the groups in the grid
- Full Collapse: Collapses all the groups in the grid
- Clear Grouping: Clears all the created groups in the grid

When a group element is available in FlexGridGroupPanel, clicking on it opens up a context menu with the following options:

- Expand: Expands the groups associated with the group element
- Collapse: Collapses the groups associated with the group element

- Clear: Clears the groups associated with the group element

However, the context menu can be customized using [ContextMenuStrip](#) property of the **C1FlexGridGroupPanel** class. This customized menu is displayed for both, FlexGridGroupPanel as well as group element.

The following code

#### Visual Basic

```
toolStripMenuItem1.Click += ContextMenuStripItems_Click
toolStripMenuItem2.Click += ContextMenuStripItems_Click
toolStripMenuItem3.Click += ContextMenuStripItems_Click
toolStripMenuItem4.Click += ContextMenuStripItems_Click
flexGroupPanel.ContextMenuStrip =contextMenuStrip1
contextMenuStrip1.BackColor = System.Drawing.Color.Aqua
contextMenuStrip1.Dock = DockStyle.Top
```

- C#

```
//Adding custom ContextMenuStrip
toolStripMenuItem1.Click += ContextMenuStripItems_Click;
toolStripMenuItem2.Click += ContextMenuStripItems_Click;
toolStripMenuItem3.Click += ContextMenuStripItems_Click;
toolStripMenuItem4.Click += ContextMenuStripItems_Click;
flexGroupPanel.ContextMenuStrip =contextMenuStrip1;
contextMenuStrip1.BackColor = System.Drawing.Color.Aqua;
contextMenuStrip1.Dock = DockStyle.Top;
```

### FlexGridGroupPanel Styling

FlexGridGroupPanel allows you to customize its appearance using [BackColor](#), [BorderStyle](#) and [BorderColor](#) properties of the C1FlexGridGroupPanel class. It also provides [Styles](#) property to customize the appearance of FlexGridGroupPanel, group element, and clear button element.

The overall appearance of FlexGridGroupPanel can be changed using the properties exposed by the [GroupPanelCommonStyle](#) class. The appearance of the group element can be changed using properties exposed by the [GroupStyle](#) class and the appearance of the clear button element can be changed using properties exposed by the [ClearButtonStyle](#) class.

#### Visual Basic

```
'Styling the FlexGridGroupPanel
flexGroupPanel.BackColor = System.Drawing.Color.Orange
flexGroupPanel.BorderStyle = BorderStyle.FixedSingle
flexGroupPanel.BorderColor = System.Drawing.Color.Green

'Setting the style for the FlexGridGroupPanel when it is disabled
flexGroupPanel.Enabled = false
flexGroupPanel.Styles.Common.Disabled.BackColor = System.Drawing.Color.Red
flexGroupPanel.Styles.Common.Disabled.ForeColor = System.Drawing.Color.Green

'Styling the C1FlexGridGroupPanel with the help of Styles property
flexGroupPanel.Styles.Common.Font = new System.Drawing.Font("Arial", 8,
System.Drawing.FontStyle.Bold)

'Styling the group element
flexGroupPanel.Styles.Group.BackColor = System.Drawing.Color.Green
flexGroupPanel.Styles.Group.ForeColor = System.Drawing.Color.Yellow
flexGroupPanel.Styles.Group.Border = 2
flexGroupPanel.Styles.Group.BorderColor = System.Drawing.Color.Black
```

```
'Setting the style for the group element when mouse is hovered over it
flexGroupPanel.Styles.Group.Hot.BackColor = System.Drawing.Color.Green
flexGroupPanel.Styles.Group.Hot.Border = 2
flexGroupPanel.Styles.Group.Hot.BorderColor = System.Drawing.Color.Black

'Styling the Clear button element
flexGroupPanel.Styles.ClearButton.BackColor = System.Drawing.Color.Yellow
flexGroupPanel.Styles.ClearButton.Image =
System.Drawing.Image.FromFile("../..\..\CloseIcon.png")
flexGroupPanel.Styles.ClearButton.Hot.BackColor = System.Drawing.Color.Red
flexGroupPanel.Styles.ClearButton.Hot.BorderColor = System.Drawing.Color.Blue
```

C#

```
//Styling the FlexGridGroupPanel
flexGroupPanel.BackColor = System.Drawing.Color.Orange;
flexGroupPanel.BorderStyle = BorderStyle.FixedSingle;
flexGroupPanel.BorderColor = System.Drawing.Color.Green;

//Setting the style for the FlexGridGroupPanel when it is disabled
flexGroupPanel.Enabled = false;
flexGroupPanel.Styles.Common.Disabled.BackColor = System.Drawing.Color.Red;
flexGroupPanel.Styles.Common.Disabled.ForeColor = System.Drawing.Color.Green;

//Styling the C1FlexGridGroupPanel with the help of Styles property
flexGroupPanel.Styles.Common.Font = new System.Drawing.Font("Arial", 8,
System.Drawing.FontStyle.Bold);

//Styling the group element
flexGroupPanel.Styles.Group.BackColor = System.Drawing.Color.Green;
flexGroupPanel.Styles.Group.ForeColor = System.Drawing.Color.Yellow;
flexGroupPanel.Styles.Group.Border = 2;
flexGroupPanel.Styles.Group.BorderColor = System.Drawing.Color.Black;

//Setting the style for the group element when mouse is hovered over it
flexGroupPanel.Styles.Group.Hot.BackColor = System.Drawing.Color.Green;
flexGroupPanel.Styles.Group.Hot.Border = 2;
flexGroupPanel.Styles.Group.Hot.BorderColor = System.Drawing.Color.Black;

//Styling the Clear button element
flexGroupPanel.Styles.ClearButton.BackColor = System.Drawing.Color.Yellow;
flexGroupPanel.Styles.ClearButton.Image =
System.Drawing.Image.FromFile("../..\..\CloseIcon.png");
flexGroupPanel.Styles.ClearButton.Hot.BackColor = System.Drawing.Color.Red;
flexGroupPanel.Styles.ClearButton.Hot.BorderColor =
System.Drawing.Color.Blue;
```

## Merging Cells

The [C1FlexGrid](#) control allows you to merge cells, making them span multiple rows or columns. This capability can be used to enhance the appearance and clarity of the data displayed on the grid. The effect of these settings is similar to the HTML `<ROWSPAN>` and `<COLSPAN>` tags.

To enable cell merging, you must do two things:

1. Set the grid's `AllowMerging` property to a value other than **None**. (The effect of each setting is explained in the reference section.)
2. If you want to merge columns, set the `AllowMerging` property to **True** for each column that you would like to merge. If you want to merge rows, set the `AllowMerging` property to **True** for each row that you would like to merge

Merging will occur if adjacent cells contain the same non-empty string. There is no method to force a pair of cells to merge. The merging is done automatically based on the cell contents. This makes it easy to provide merged views of sorted data, where values in adjacent rows present repeated data.

Cell merging has several possible uses. For example, you can use it to create merged table headers, merged data views, or grids where the text spills into adjacent columns.

## Merged Table Headers

To create merged table headers, you must start by setting the grid's `AllowMerging` property to **FixedOnly**. Then, designate the rows and columns that you want to merge by setting the row and column's `AllowMerging` properties. Finally, assign the text to the header cells so that the cells you want to merge have the same contents.

The code below shows an example:

### To write code in Visual Basic

#### Visual Basic

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs)
    Handles MyBase.Load
    Dim i%

    ' Initialize the control.
    _flex.Styles.Normal.WordWrap = True
    _flex.Cols.Count = 9
    _flex.Rows.Fixed = 2
    _flex.AllowMerging = C1.Win.C1FlexGrid.AllowMergingEnum.FixedOnly

    ' Create row headers.
    _flex.Rows(0).AllowMerging = True

    ' Merge the four cells with same contents.
    Dim rng As C1.Win.C1FlexGrid.CellRange = _flex.GetCellRange(0, 1, 0, 4)
    rng.Data = "North"

    ' Merge the four cells with same contents.
    rng = _flex.GetCellRange(0, 5, 0, 8)
    rng.Data = "South"
    For i = 1 To 4
        _flex(1, i) = "Qtr " & i
        _flex(1, i + 4) = "Qtr " & i
    Next

    ' Create the column header.
    _flex.Cols(0).AllowMerging = True
```

```
' Merge the two cells with same contents.
rng = _flex.GetCellRange(0, 0, 1, 0)
rng.Data = "Sales by Product"

' Align and autosize the cells.
_flex.Styles.Fixed.TextAlign = C1.Win.C1FlexGrid.TextAlignEnum.CenterCenter
_flex.AutoSizeCols(1, _flex.Cols.Count - 1, 10)
End Sub
```

### To write code in C#

```
C#
private void Form1_Load(System.Object sender, System.EventArgs e)
{
    int i;

    // Initialize the control.
    _flex.Styles.Normal.WordWrap = true;
    _flex.Cols.Count = 9;
    _flex.Rows.Fixed = 2;
    _flex.AllowMerging = C1.Win.C1FlexGrid.AllowMergingEnum.FixedOnly;

    // Create row headers.
    _flex.Rows[0].AllowMerging = true;

    // Merge the four cells with same contents.
    C1.Win.C1FlexGrid.CellRange rng = _flex.GetCellRange(0, 1, 0, 4);
    rng.Data = "North";

    // Merge the four cells with same contents.
    rng = _flex.GetCellRange(0, 5, 0, 8);
    rng.Data = "South";
    for ( i = 1 ; i <= 4; i++)
    {
        _flex[1, i] = "Qtr " + i;
        _flex[1, i + 4] = "Qtr " + i;
    }

    // Create the column header.
    _flex.Cols[0].AllowMerging = true;

    // Merge the two cells with same contents.
    rng = _flex.GetCellRange(0, 0, 1, 0);
    rng.Data = "Sales by Product";

    // Align and autosize the cells.
    _flex.Styles.Fixed.TextAlign = C1.Win.C1FlexGrid.TextAlignEnum.CenterCenter;
    _flex.AutoSizeCols(1, _flex.Cols.Count - 1, 10);
}
```

This is the result:



Sales by Product	North				South			
	Qtr 1	Qtr 2	Qtr 3	Qtr 4	Qtr 1	Qtr 2	Qtr 3	Qtr 4

## Merged Data Views

Cell merging works the same way when the grid is bound to a data source. The code below shows an example for a grid bound to a data source at design time. For more information on binding to a data source, see [Binding to a Data Source](#).

### To write code in Visual Basic

#### Visual Basic

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs)
    Handles MyBase.Load
        Dim i%

        ' Set up cell merging.
        _flex.AllowMerging = C1.Win.C1FlexGrid.AllowMergingEnum.RestrictCols
        For i = _flex.Cols.Fixed To _flex.Cols.Count - 1
            _flex.Cols(i).AllowMerging = True
        Next
    End Sub
```

### To write code in C#

#### C#


```
private void Form1_Load( System.object sender, System.EventArgs e)
{
    int i;

    // Set up cell merging.
    _flex.AllowMerging = C1.Win.C1FlexGrid.AllowMergingEnum.RestrictCols;
    for (int i = _flex.Cols.Fixed; i <= _flex.Cols.Count - 1; i++)
    {
        _flex.Cols(i).AllowMerging = true;
    }
}
```

This is the result:

Country	City	CompanyName	ContactName
Argentina	Buenos Aires	Cactus Comidas para llevar	Patricio Simpson
		Océano Atlántico Ltda.	Yvonne Moncada
		Rancho grande	Sergio Gutiérrez
Austria	Graz	Ernst Handel	Roland Mendel
	Salzburg	Piccolo und mehr	Georg Pippis
Belgium	Bruxelles	Maison Dewey	Catherine Dewey
	Charleroi	Suprêmes délices	Pascale Cartrain
Brazil	Campinas	Gourmet Lanchonetes	André Fonseca
		Wellington Importadora	Paula Parente
	Rio de Janeiro	Hanari Carnes	Mario Pontes
		Que Delícia	Bernardo Batista
		Ricardo Adocicados	Janete Limeira
	São Paulo	Comércio Mineiro	Pedro Afonso
		Familia Arquibaldo	Aria Cruz
		Queen Cozinha	Lúcia Carvalho
		Tradição Hipermercados	Anabela Domingues

Notice how merging the cells has the effect of visually grouping the data and making the information on the table easier to understand.

 **Note:** For an example of displaying merged data views with [C1FlexGrid](#), see the **CellMerging** sample on [GrapeCity website](#).

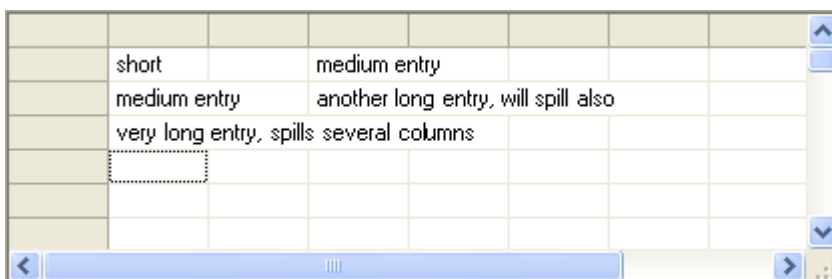
## Spilling Text

The [AllowMerging](#) property has two settings that operate differently from the others that do not require you to set the AllowMerging property on specific rows and columns.

## Spill Setting

The **Spill** setting causes text that is too long to fit in a cell to spill into empty adjacent cells. The resulting behavior is similar to the one in Microsoft Excel. If you type a long entry into a cell and the adjacent cell is empty, the entry will spill to occupy as much room as needed.

For example, the picture below shows what a grid might look like when AllowMerging is set to **Spill** and the user types entries of varying lengths:



## Nodes Setting

The **Nodes** setting is similar to **Spill** but only applies to outline nodes. This setting is useful when data is organized into groups, and the node rows contain information in a format different from the data rows.

For example, the picture below shows what a grid might look like when the data is grouped and summarized using the `Subtotal` method, and then `AllowMerging` is set to **Nodes**:

Product	Region	Associate	Sales
<b>Grand Total</b>			<b>\$646,337</b>
<b>Total Drums</b>			<b>\$297,538</b>
<b>Total North</b>			<b>\$182,157</b>
Drums	North	Paul	\$86,706
Drums	North	Paul	\$93,497
Drums	North	Sylvia	\$1,954
<b>Total East</b>			<b>\$115,381</b>
Drums	East	Sylvia	\$22,785
Drums	East	Sylvia	\$92,596
<b>Total Flutes</b>			<b>\$348,799</b>
<b>Total East</b>			<b>\$169,579</b>
Flutes	East	Sylvia	\$64,006
Flutes	East	John	\$33,011
Flutes	East	John	\$72,562

This image is similar to the one in the [Creating Subtotals](#) topic. The difference is that the subtotal rows (nodes) now spill onto empty adjacent cells, improving the appearance of the grid.

## Custom Merging

You can customize the default merging behavior in two ways:

- Assign a custom `IComparer` class to the `CustomComparer` property.

By default, the grid will merge adjacent cells that contain the same non-null value. String comparisons are case-sensitive and blanks are included.

If you wanted the grid to merge cells using a case-insensitive comparison and trimming blanks, you could write a custom class that implements `IComparer` and assign it to the `CustomComparer` property.

- Write a new class that derives from the `C1FlexGrid` and override the `GetMergedRange` virtual method, providing your own custom merging logic.

**Note:** For samples that show how to implement custom merging logic, see the **CustomMerge**, **CustomMerge2**, **CustomMerge3**, and **CustomMerge4** samples on [GrapeCity website](#).

## Outlining and Summarizing Data

The `C1FlexGrid` control has methods and properties that allow you to summarize data and display it in a hierarchical manner. To summarize data and add aggregate values, use the `Subtotal` method. To display hierarchical views of the data, use the `Tree` property.

## Creating Subtotals

The `C1FlexGrid.Subtotal` method adds subtotal rows that contain aggregate data for the regular (non-subtotal) rows.

Subtotal supports hierarchical aggregates. For example, if your grid contains sales data, you may Subtotal to get aggregate sales figures by Product, Region, and Salesperson. The code below illustrates this:

#### To write code in Visual Basic

##### Visual Basic

```
Private Sub ShowTotals()  
  
    ' Show OutlineBar on column 0.  
    _flex.Tree.Column = 0  
    _flex.Tree.Style = TreeStyleFlags.Simple  
  
    ' Clear existing subtotals.  
    _flex.Subtotal(AggregateEnum.Clear)  
  
    ' Get a Grand total (use -1 instead of column index).  
    _flex.Subtotal(AggregateEnum.Sum, -1, -1, 3, "Grand Total")  
  
    ' Total per Product (column 0).  
    _flex.Subtotal(AggregateEnum.Sum, 0, 0, 3, "Total {0}")  
  
    ' Total per Region (column 1).  
    _flex.Subtotal(AggregateEnum.Sum, 1, 1, 3, "Total {0}")  
  
    ' Size column widths based on content.  
    _flex.AutoSizeCols()  
End Sub
```

#### To write code in C#

##### C#

```
private void ShowTotals()  
{  
    // Show OutlineBar on column 0.  
    _flex.Tree.Column = 0;  
    _flex.Tree.Style = TreeStyleFlags.Simple;  
  
    // Clear existing subtotals.  
    _flex.Subtotal(AggregateEnum.Clear);  
  
    // Get a Grand total (use -1 instead of column index).  
    _flex.Subtotal(AggregateEnum.Sum, -1, -1, 3, "Grand Total");  
  
    // Total per Product (column 0).  
    _flex.Subtotal(AggregateEnum.Sum, 0, 0, 3, "Total {0}");  
  
    // Total per Region (column 1).  
    _flex.Subtotal(AggregateEnum.Sum, 1, 1, 3, "Total {0}");  
  
    // Size column widths based on content.  
    _flex.AutoSizeCols();  
}
```

```
}
```

When the `C1FlexGrid.Subtotal` method adds rows with the aggregate information, it automatically assigns subtotal styles to the new rows (there are built-in styles for 5 levels of subtotals). You can customize the appearance of the subtotal rows by changing the properties of the outline styles in the designer with the **Style Editor** or with code. For example:

#### To write code in Visual Basic

##### Visual Basic

```
' Set styles for subtotals.
Dim cs As C1.Win.C1FlexGrid.CellStyle
cs = _flex.Styles(C1.Win.C1FlexGrid.CellStyleEnum.GrandTotal)
cs.BackColor = Color.Black
cs.ForeColor = Color.White
cs.Font = New Font(Font, FontStyle.Bold)

cs = _flex.Styles(C1.Win.C1FlexGrid.CellStyleEnum.Subtotal0)
cs.BackColor = Color.DarkRed
cs.ForeColor = Color.White
cs.Font = New Font(Font, FontStyle.Bold)

cs = _flex.Styles(C1.Win.C1FlexGrid.CellStyleEnum.Subtotal1)
cs.BackColor = Color.DarkBlue
cs.ForeColor = Color.White
```

#### To write code in C#

##### C#

```
// Set styles for subtotals.
CellStyle cs;
cs = _flex.Styles[CellStyleEnum.GrandTotal];
cs.BackColor = Color.Black;
cs.ForeColor = Color.White;
cs.Font = new Font(Font, FontStyle.Bold);

cs = _flex.Styles[CellStyleEnum.Subtotal0];
cs.BackColor = Color.DarkRed;
cs.ForeColor = Color.White;
cs.Font = new Font(Font, FontStyle.Bold);

cs = _flex.Styles[CellStyleEnum.Subtotal1];
cs.BackColor = Color.DarkBlue;
cs.ForeColor = Color.White;
```

After executing this code, the grid would look like this:

Product	Region	Associate	Sales
<b>Grand Total</b>			<b>\$592,944</b>
<b>Total Drums</b>			<b>\$351,331</b>
<b>Total North</b>			<b>\$196,640</b>
Drums	North	Paul	\$74,257
Drums	North	Paul	\$77,395
Drums	North	Sylvia	\$44,988
<b>Total East</b>			<b>\$154,691</b>
Drums	East	Sylvia	\$93,033
Drums	East	Sylvia	\$61,658
<b>Total Flutes</b>			<b>\$241,613</b>
<b>Total East</b>			<b>\$142,682</b>
Flutes	East	Sylvia	\$87,030
Flutes	East	John	\$34,780
Flutes	East	John	\$20,872

The Grand Total row contains the total sales for all products, regions, and sales personnel. It was created using a `-1` for the `groupOn` parameter in the call to the `C1FlexGrid.Subtotal` method. The other subtotals show total sales by product and region. They were created using a values `0` and `1` for the `groupOn` parameter.

For information on grouping, please refer [Grouping](#).

You may also calculate aggregates other than sums (for example, averages or percentages), and calculate several aggregates for each row (for example, gross and net sales).

Subtotal rows created by the **Subtotal** method differ from regular rows in three aspects:

1. Subtotal rows can be automatically removed by invoking the **Subtotal** method with the `flexSTClear` parameter. This is useful to provide dynamic views of the data, where the user may move columns and re-sort the data, making it necessary to recalculate the subtotals.
2. Subtotal rows can be used as nodes in an outline, allowing you to collapse and expand groups of rows to present an overview of the data or to reveal its details. To see the outline tree, you need to set the `Column` and `Tree.Style` properties to define the position and appearance of the outline tree.
3. Subtotal rows can be treated as nodes in a tree. You can get a `Node` object for any subtotal row through the `Node` property.
4. When the grid is bound to a data source, the subtotal rows do not correspond to actual data. If you move the cursor in the data source, subtotal rows will be skipped in the grid.

The outline tree allows users to collapse and expand sections of the grid by clicking on the nodes. You can use outline trees to display many types of information, not only aggregates. The next topic shows how you can create a custom outline tree to display directory information.

## Creating Custom Subtotal

FlexGrid can display subtotals on grouped rows using `Subtotal` method. In addition, it allows you to create custom subtotal using custom expressions as subtotals in groups along with aggregates. You can create custom group expressions for a column using `GroupExpressions` property of the `Column` class.

The following image shows the calculated subtotal in group:

ID	FirstName	LastName	Department	Age	JoiningDate	Attendance	Expression
Department: IT				43	27-06-2019		6
1	John	Wayne	IT	35	01-12-2007	5	35
1	Maximus	Decimus	IT	33	15-05-2011	6	33
2	Alexander	Conklin	IT	42	22-06-2004	3.93	84
3	William	Boume	IT	66	15-04-2015	8.94	198
4	Jacob	Lee	IT	26	11-08-2017	4.98	104
5	Mike	Boume	IT	56	21-01-1999	7.81	280
Department: Ad				31	27-06-2019		4
6	Fuller	Brown	Administration	36	07-08-2012	3.99	216
7	Miller	Wayne	Administration	42	19-07-2014	4.75	294
8	Sam	Conklin	Administration	22	15-06-2018	1.97	176
9	Mark	Boume	Administration	24	15-06-2016	6	216

The following code can be used to use custom expressions as subtotals for creating custom subtotals. In this example, we have used a list of employees as a data source for the FlexGrid control.

## Visual Basic

```
Dim grps As List(Of GroupDescription) = New List(Of GroupDescription)()
Dim grp As GroupDescription = New GroupDescription("Department",
ListSortDirection.Descending, True)
grps.Add(grp)
c1FlexGrid1.GroupDescriptions = grps

Dim column = c1FlexGrid1.Cols.Add()
column.Name = "Expression"
column.DataType = GetType(Object)
column.Caption = "Expression"
column.AllowEditing = False
column.Expression = "[ID] * [Age]"

'GroupExpression implementation
c1FlexGrid1.Cols("ID").GroupExpression = "=Count([ID])"
c1FlexGrid1.Cols("Age").GroupExpression = "=Average([Age])"
c1FlexGrid1.Cols("JoiningDate").GroupExpression = "=Now()"
c1FlexGrid1.Cols("Expression").GroupExpression =
"=iif(10<5,Sum([Age]),Count([Age]))"
```

## C#

```
List<GroupDescription> grps = new List<GroupDescription>();
GroupDescription grp = new GroupDescription("Department",
ListSortDirection.Descending, true);
grps.Add(grp);
c1FlexGrid1.GroupDescriptions = grps;

var column = c1FlexGrid1.Cols.Add();
column.Name = "Expression";
column.DataType = typeof(object);
column.Caption = "Expression";
column.AllowEditing = false;
```

```
column.Expression = "[ID] * [Age]";

//GroupExpression implementation
c1FlexGrid1.Cols["ID"].GroupExpression = "=Count([ID])";
c1FlexGrid1.Cols["Age"].GroupExpression = "=Average([Age])";
c1FlexGrid1.Cols["JoiningDate"].GroupExpression = "=Now()";
c1FlexGrid1.Cols["Expression"].GroupExpression =
    "=iif(10<5,Sum([Age]),Count([Age]))";
```

## Creating Custom Trees

To create outline trees without using the **Subtotal** method, you need to follow these steps:

1. Add rows to the grid.
2. Turn some rows into outline nodes by setting their `IsNode` property to **True**.
3. Get the `Node` object for each node row and set its `Level` property to define the node's position in the tree hierarchy. Higher values mean the node is deeper (more indented) into the outline tree.

For example, the code below creates a directory tree:

### To write code in Visual Basic

#### Visual Basic

```
' add these Imports statements at the top of the for.
Imports System.IO
Imports Cl.Win.C1FlexGrid

Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs)
    Handles MyBase.Load

    ' Initialize grid layout.
    _flex.Cols.Fixed = 0
    _flex.Cols.Count = 1
    _flex.Rows.Count = 1
    _flex.ExtendLastCol = True
    _flex.Styles.Normal.TextAlign = TextAlignEnum.LeftCenter
    _flex.Styles.Normal.Border.Style = BorderStyleEnum.None

    ' Show outline tree.
    _flex.Tree.Column = 0
    _flex.Tree.Style = TreeStyleFlags.SimpleLeaf
    _flex.Tree.LineColor = Color.DarkBlue

    ' Populate the grid.
    AddDirectory("c:\", 0)
End Sub
```

### To write code in C#

#### C#

```
// add these using statements at the top of the for.
```



```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.IO;
using Cl.Win.ClFlexGrid;

private void Form1_Load(object sender, EventArgs e)
{
    // Initialize grid layout.
    _flex.Cols.Fixed = 0;
    _flex.Cols.Count = 1;
    _flex.Rows.Count = 1;
    _flex.ExtendLastCol = true;
    _flex.Styles.Normal.TextAlign = TextAlignEnum.LeftCenter;
    _flex.Styles.Normal.Border.Style = BorderStyleEnum.None;

    // Initialize outline tree.
    _flex.Tree.Column = 0;
    _flex.Tree.Style = TreeStyleFlags.SimpleLeaf;
    _flex.Tree.LineColor = Color.DarkBlue;

    // Populate the grid.
    AddDirectory(@"c:\\", 0);
}
```

The code above initializes the grid layout and calls the **AddDirectory** routine that does the job of populating the grid and setting up the tree structure:

#### To write code in Visual Basic

##### Visual Basic

```
Private Sub AddDirectory(ByVal dir As String, ByVal level As Integer)

    ' Add this directory.
    Dim thisDir As String
    thisDir = Path.GetFileName(dir)
    If thisDir.Length = 0 Then thisDir = dir
    _flex.AddItem(thisDir)

    ' Make this new row a node.
    Dim row As Row
    row = _flex.Rows(_flex.Rows.Count - 1)
    row.IsNode = True

    ' Set the node level.
    Dim nd As Node
    nd = row.Node
```

```

nd.Level = level

' Add files in this directory.
Dim file As String, cnt As Integer, r As Row
cnt = 0
For Each file In Directory.GetFiles(dir)
    _flex.AddItem(Path.GetFileName(file).ToLower())
        r = _flex.Rows(_flex.Rows.Count - 1)
        r.IsNode = True
        r.Node.Level = level + 1
    cnt = cnt + 1
    If cnt > 10 Then Exit For
Next

' Add subdirectories (up to level 4).
If level <= 4 Then
    Dim subdir As String
    cnt = 0
    For Each subdir In Directory.GetDirectories(dir)
        AddDirectory(subdir, level + 1)
        cnt = cnt + 1
        If cnt > 10 Then Exit For
    Next
End If
End Sub

```

### To write code in C#

```

C#
private void AddDirectory(string dir, int level)
{
    // add this director.
    string thisDir = Path.GetFileName(dir);
    if (thisDir.Length == 0) { thisDir = dir; }
    _flex.AddItem(thisDir);
    //make this new row a node
    Row row = _flex.Rows[_flex.Rows.Count - 1];
    row.IsNode = true;
    //set node level
    Node nd = row.Node;
    nd.Level = level;
    // add files in this director.
    int cnt = 0;
    Row r;

    foreach (string file in Directory.GetFiles(dir))
    {
        _flex.AddItem(Path.GetFileName(file).ToLower());
        //mark the row without child row as node
        r = _flex.Rows[_flex.Rows.Count - 1];
        r.IsNode = true;
    }
}

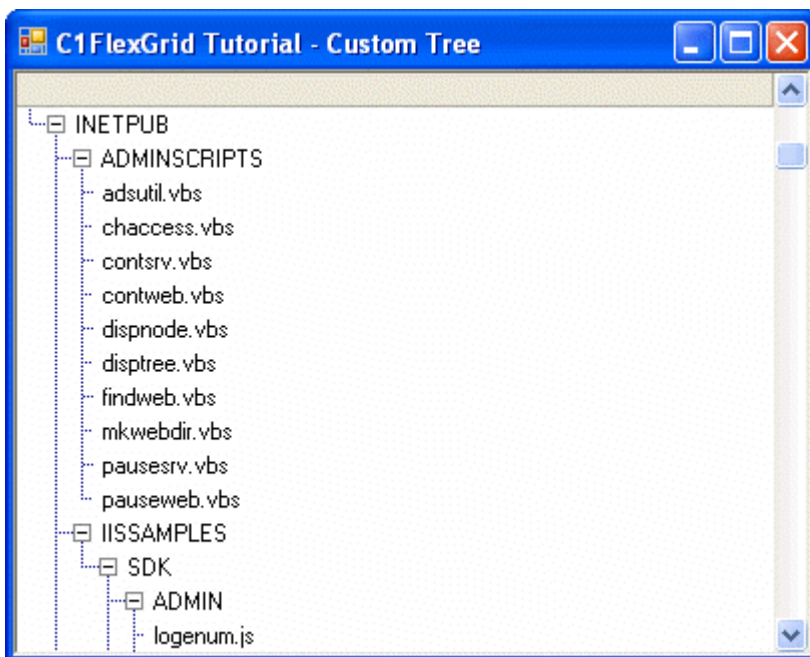
```

```
        r.Node.Level = level + 1;
        cnt = cnt + 1;
        if (cnt > 10) break;
    }

    // add subdirectories (up to level 4.
    if (level <= 4)
    {
        cnt = 0;
        foreach (string subdir in Directory.GetDirectories(dir))
        {
            AddDirectory(subdir, level + 1);
            cnt = cnt + 1;
            if (cnt > 10) break;
        }
    }
}
```

**AddDirectory** is a recursive routine that traverses the current directory and all its subdirectories. In this example, the tree size is limited to four directory levels in order to save time. In a real application, the routine should be changed to populate tree branches only when they are expanded (see the [FlexGrid for WinForms Tutorials](#)).

This code creates a grid that looks like this:



## Creating Outlines and Trees with the C1FlexGrid Control

One of the unique and popular features of the [C1FlexGrid](#) control is the ability to add hierarchical grouping to regular unstructured data.

To achieve this, the [C1FlexGrid](#) introduces the concept of **Node** rows. Node rows do not contain regular data. Instead, they act as headers under which similar data is grouped, exactly like nodes in a regular **TreeView** control. Like nodes in a **TreeView** control, node rows can be collapsed and expanded, hiding or showing the data they contain. Also like

nodes in a **TreeView** control, node rows have a **Level** property that defines the node hierarchy. Lower level nodes contain higher level nodes.

For example, suppose you had a grid showing customer name, country, city, and sales amounts. This typical grid would normally look like this:

	Country	City	SalesPerson	Quantity	ExtendedPrice
	Argentina	Buenos Aires	Andrew Fuller	7	322.00
	Argentina	Buenos Aires	Andrew Fuller	20	155.00
	Argentina	Buenos Aires	Anne Dodsworth	1	12.50
	Argentina	Buenos Aires	Anne Dodsworth	2	527.00
	Argentina	Buenos Aires	Anne Dodsworth	5	405.00
	Argentina	Buenos Aires	Janet Leverling	12	96.00
	Argentina	Buenos Aires	Janet Leverling	12	223.20
	Argentina	Buenos Aires	Laura Callahan	3	54.00
	Argentina	Buenos Aires	Laura Callahan	6	75.00

All the information is there, but it's hard to see the total sales for each country or customer. You could use the **C1FlexGrid**'s outlining features to group the data by country (level 0), then by city within each country (level 1), then by customer within each city (level 2). Here is the same grid with after adding the outline:

	SalesPerson	Quantity	ExtendedPrice
[-]	<b>Argentina</b>	<b>339</b>	<b>8,119.10</b>
[+]	Buenos Aires	339	8,119.10
[-]	<b>Austria</b>	<b>5167</b>	<b>128,003.83</b>
[+]	Graz	4543	104,874.97
[+]	Salzburg	624	23,128.86
[-]	<b>Belgium</b>	<b>1392</b>	<b>33,824.85</b>
[-]	Bruxelles	320	9,736.07
	Anne Dodsworth	6	99.19
	Anne Dodsworth	10	184.00

This grid shows the same information as the previous one (it is bound to the same data source), but it adds a tree where each node contains a summary of the data below it. Nodes can be collapsed to show only the summary, or expanded to show the detail. Note that each node row can show summaries for more than one column (in this case, total units sold and total amount).

In this article, we will walk you through the process of turning a regular grid into a richer outline grid.

## Loading the Data

Loading data into an outline grid is exactly the same as loading it into a regular grid. If your data source is available at design time, you can use the Visual Studio Property Window to set the grid's **DataSource** property and bind the grid to the data without writing any code.

If the data source is not available at design time, you can set the grid's **DataSource** property in code. The data binding code typically looks like this:

### To write code in C#

```
C#
public Form1()
{
    InitializeComponent();

    // get data
    var fields = @"
        Country,
        City,
        SalesPerson,
        Quantity,
        ExtendedPrice";
    var sql = string.Format("SELECT {0} FROM Invoices ORDER BY {0}", fields);
    var da = new OleDbDataAdapter(sql, GetConnectionString());
    da.Fill(_dt);

    // bind grid to data
    this._flex.DataSource = _dt;

    // format ExtendedPrice column
    _flex.Cols["ExtendedPrice"].Format = "n2";
}
```

The code uses an **OleDbDataAdapter** to fill a **DataTable** with data, then and then assigns the **DataTable** to the grid's **DataSource** property.

After running this code, you would get the "regular grid" shown in the first image. To turn this regular grid into the outline grid shown in the second image, we need to insert the node rows that make up the outline.

## Creating Node Rows

Node rows are almost identical to regular rows, except for the following:

- Node rows are not data bound. When the grid is bound to a data source, each regular row corresponds to an item in the data source. Node rows do not. Instead, they exist to group regular rows that contain similar data.
- Node rows can be collapsed or expanded. When a node row is collapsed, all its data and child nodes are hidden. If the outline tree is visible, users can collapse and expand nodes using the mouse or the keyboard. If the outline tree is not visible, then nodes can only be expanded or collapsed using code.

To determine whether a row is a node or not, you can use the [IsNode](#) property:

### To write code in C#

```
C#
var row = _flex.Rows[rowIndex];
if (row.IsNode)
{
    // row is a nod.
    var node = row.Node;
    DoSomethingWithTheNode (node);
}
else
```

```
{  
    // this row is not a nod.  
}
```

Node rows can be created in three ways:

1. Use the **Rows.InsertNode** method. This will insert a new node row at a specified index. Once the node row has been created, you can use it like you would any other row (set the data for each column, apply styles, etc.). This is the 'low-level' way of inserting totals and building outlines. It gives the most control and flexibility and is demonstrated below.
2. Use the **Subtotal** method. This method scans the entire grid and automatically inserts node rows with optional subtotals at locations where the grid data changes. This is the 'high-level' way of inserting totals and building outlines. It requires very little code, but makes some assumptions about how the data is structured on the grid and what the outline should look like.
3. If the grid is unbound, then you can turn regular rows into node rows by setting the **IsNode** property to true. Note that this only works when the grid is unbound. Trying to turn a regular data bound row into a node will cause the grid to throw an exception.

The code below shows how you could implement a **GroupBy** method that inserts node rows grouping identical values on a given column.

**To write code in C#**

```
C#  
  
// group on a given column inserting nodes of a given leve.  
void GroupBy(string columnName, int level)  
{  
    object current = null;  
    for (int r = _flex.Rows.Fixed; r < _flex.Rows.Count; r++)  
    {  
        if (!_flex.Rows[r].IsNode)  
        {  
            var value = _flex[r, columnName];  
            if (!object.Equals(value, current))  
            {  
                // value changed: insert nod.  
                _flex.Rows.InsertNode(r, level);  
  
                // show group name in first scrollable colum.  
                _flex[r, _flex.Cols.Fixed] = value;  
  
                // update current valu.  
                current = value;  
            }  
        }  
    }  
}
```

The code scans all the columns, skipping existing node rows (so it can be called to add several levels of nodes), and keeps track of the current value for the column being grouped on. When the current value changes, a node row is inserted showing the new group name in the first scrollable column.

Back to our example, you could use this method to create a two-level outline by calling:

**To write code in C#**

```
C#  
  
void _btnGroupCountryCity_Click(object sender, EventArgs e)  
{  
    GroupBy("Country", 0);  
    GroupBy("City", 1);  
}
```

Very simple, but there are some caveats. First, the method assumes that the data is sorted according to the outline structure. In this example, if the data were sorted by **SalesPerson** instead of by **Country**, the outline would have several level-0 nodes for each country, which probably is not what you want.

Also, the **GroupBy** method may insert many rows, which would cause the grid to flicker. To avoid this, you would normally set the **Redraw** property to false before making the updates and set it back to true when done.

To handle these issues, the code that creates the outline should be re-written as follows:

**To write code in C#**

```
C#  
  
void _btnGroupCountryCity_Click(object sender, EventArgs e)  
{  
    // suspend redrawing while updating.  
    using (new DeferRefresh(_flex))  
    {  
        // restore original sort (by Country, City, etc..  
        ResetBinding();  
  
        // group by Country, City.  
        GroupBy("Country", 0);  
        GroupBy("City", 1);  
    }  
}
```

The **DeferRefresh** class is a simple utility that sets the grid's **Redraw** property to false and restores its original value when it is disposed. This ensures that **Redraw** is properly restored even when exceptions happen during the updates. Here is the implementation of the **DeferRefresh** class:

**To write code in C#**

```
C#  
  
// Utility class used to encapsulate grid lengthy operations in a Redraw block.  
// This avoids flicker and ensures the Redraw property is reset properly in case.  
// an exception is thrown during the operation.  
class DeferRefresh : IDisposable  
{  
    C1FlexGrid _grid;  
    bool _redraw;  
    public DeferRefresh(C1FlexGrid grid)  
    {  
        _grid = grid;  
        _redraw = grid.Redraw;  
    }  
}
```

```
        grid.Redraw = false;
    }
    public void Dispose()
    {
        _grid.Redraw = _redraw;
    }
}
```

The **BindGrid** method ensures that the grid is sorted in the order required by our outline structure. In our example, the sort order is by **Country**, **City**, and **SalesPerson**. The code looks like this:

#### To write code in C#

```
C#
// unbind and re-bind grid in order to reset everythin.
void ResetBinding()
{
    // unbind gri.
    _flex.DataSource = null;

    // reset any custom sortin.
    _dt.DefaultView.Sort = string.Empty;

    // re-bind gri.
    _flex.DataSource = _dt;

    // format ExtendedPrice colum.
    _flex.Cols["ExtendedPrice"].Format = "n2";

    // auto-size the columns to fit their conten.
    _flex.AutoSizeCols();
}
```

If you run this code now, you will notice that the node rows are created as expected, but the outline tree is not visible, so you can't expand and collapse the nodes. The outline tree is described in the next section.

## Outline Tree

The outline tree is very similar to the one you see in a regular **TreeView** control. It shows an indented structure with collapse/expand icons next to each node row so the user can expand and collapse the outline to see the desired level of detail.

The outline tree can be displayed in any column, defined by the **Tree.Column** property. By default, this property is set to -1, which causes the tree not to be displayed at all. To show the outline tree in the example given above, you would use this code:

#### To write code in C#

```
C#
void _btnTreeCountryCity_Click(object sender, EventArgs e)
{
    using (new DeferRefresh(_flex))
```



```
{
    // group by country and city as before
    _btnGroupCountryCity_Click(this, EventArgs.Empty);

    // show outline tree
    _flex.Tree.Column = 0;

    // autosize to accommodate tree
    _flex.AutoSizeCol(_flex.Tree.Column);

    // collapse detail nodes
    _flex.Tree.Show(1);
}
}
```

The code calls the previous method to build the outline, then and then sets the **Tree.Column** property to zero in order to show the outline tree in the first column. It also calls the `C1FlexGrid.AutoSizeCol` method to ensure that the column is wide enough to accommodate the outline tree. Finally, it calls the **Tree.Show** method to display all level-0 nodes (cities in this case) and hide all the detail.

The **Tree** property returns a reference to a `GridTree` object that exposes several methods and properties used to customize the outline tree. The main ones are listed below:

- **Column:** Gets or sets the index of the column that contains the outline tree. Setting this property to -1 causes the outline tree to be hidden from the users.
- **Indent:** Gets or sets the indent, in pixels, between adjacent node levels. Higher indent levels cause the tree to become wider.
- **Style:** Gets or sets the type of outline tree to display. Use this property to determine whether the tree should include a button bar at the top to allow users to collapse/expand the entire tree, whether lines and/or symbols should be displayed, and whether lines should be displayed connecting the tree to data rows as well as node rows.
- **LineColor:** Gets or sets the color of the tree's connecting lines.
- **LineStyle:** Gets or sets the style of the tree's connecting lines.

For example, by changing the code above to include these two lines:

#### To write code in C#

```
C#
// show outline tree
_flex.Tree.Column = 0;
_flex.Tree.Style = TreeStyleFlags.CompleteLeaf;
_flex.Tree.LineColor = Color.White;
_flex.Tree.Indent = 30;
```

The outline tree would change as follows:

1	2 *	SalesPerson	Quantity	ExtendedPrice
[-]		<b>Argentina</b>	<b>339</b>	<b>8,119.10</b>
	[+]	Buenos Aires	339	8,119.10
[-]		<b>Austria</b>	<b>5167</b>	<b>128,003.83</b>
	[+]	Graz	4543	104,874.97
	[+]	Salzburg	624	23,128.86
[-]		<b>Belgium</b>	<b>1392</b>	<b>33,824.85</b>
	[-]	Bruxelles	320	9,736.07
		Anne Dodsworth	6	99.19
		Anne Dodsworth	10	184.00

Notice the buttons labeled "1", "2", and "\*" on the top left cell. Clicking these buttons would cause the entire tree to collapse or expand to the corresponding level. Also notice the much wider indentation and the lines connecting the tree to regular rows ("Anne Dodsworth") as well as to node rows.

## Adding Subtotals

So far we have covered the creation of node rows and outline trees. To make the outlines really useful, however, the node rows should include summary information for the data they contain.

If you create an outline tree using the [Subtotal](#) method, then the subtotals are added automatically. This will be described in a later section.

If you created the outline tree using the **Rows.InsertNode** method as described above, then you should use the [Aggregate](#) method to calculate the subtotals for each group of rows and insert the result directly into the node rows.

The `C1FlexGrid.Subtotal` method listed below shows how to do this:

### To write code in C#

```
C#
// add subtotals to each node at a given level
void AddSubtotals(int level, string colName)
{
    // get column we are going to total on
    int colIndex = _flex.Cols.IndexOf(colName);

    // scan rows looking for nodes at the right level
    for (int r = _flex.Rows.Fixed; r < _flex.Rows.Count; r++)
    {
        if (_flex.Rows[r].IsNode)
        {
            var node = _flex.Rows[r].Node;
            if (node.Level == level)
            {
                // found a node, calculate the sum of extended price
                var range = node.GetCellRange();
                var sum = _flex.Aggregate(AggregateEnum.Sum,
                    range.r1, colIndex, range.r2, colIndex,
                    AggregateFlags.ExcludeNodes);

                // show the sum on the grid
            }
        }
    }
}
```

```

        // (will use the column format automatically)
        _flex[r, colIndex] = sum;
    }
}
}
}
}

```

The **AddSubtotals** method scans the grid rows looking for node rows. When a node row of the desired level is found, the method uses the [GetCellRange](#) method to retrieve the node's child rows. Then it uses the `C1FlexGrid.Aggregate` method to calculate the sum of the values on the target column over the entire range. The call to **Aggregate** includes the **ExcludeNodes** flag to avoid double-counting existing nodes. Once the subtotal has been calculated, it is assigned to the node row's cell with the usual **\_flex[row, col]** indexer.

Note that this does not affect the data source in any way, since node rows are not bound to the data.

Note also that the method can be used to add multiple totals to each node row. In this example, we will add totals for the **Quantity** and **ExtendedPrice** columns. In addition to sums, you could add other aggregates such as average, maximum, minimum, etc.

We can now use this method to create a complete outline, with node rows, outline tree, and subtotals:

#### To write code in C#

C#

```

void _btnTreeCountryCity_Click(object sender, EventArgs e)
{
    using (new DeferRefresh(_flex))
    {
        // restore original sort (by Country, City, SalesPerson)
        ResetBinding();

        // group by Country, City
        GroupBy("Country", 0); // group by country (level 0)
        GroupBy("City", 1);    // group by city (level 1)

        // add totals per Country, City
        AddSubtotals(0, "ExtendedPrice"); // extended price per country (level 0)
        AddSubtotals(0, "Quantity");      // quantity per country (level 0)
        AddSubtotals(1, "ExtendedPrice"); // extended price per city (level 1)
        AddSubtotals(1, "Quantity");      // quantity per city (level 1)

        // show outline tree
        _flex.Tree.Column = 0;
        _flex.AutoSizeCol(_flex.Tree.Column);
        _flex.Tree.Show(1);
    }
}

```

If you run the project now, you will see a tree with node rows that show the total quantity and amount sold for each country and city. This is very nice, but there is a little problem. If you expand any of the node rows, you will see a lot of duplicate values. All rows under a given city node have the same country and city:>

	Country	City	SalesPerson	Quantity	ExtendedPrice
[-]	<b>Argentina</b>			<b>339</b>	<b>8,119.10</b>
[+]	Buenos Aires			339	8,119.10
[-]	<b>Austria</b>			<b>5167</b>	<b>128,003.83</b>
[+]	Graz			4543	104,874.97
[+]	Salzburg			624	23,128.86
[-]	<b>Belgium</b>			<b>1392</b>	<b>33,824.85</b>
[-]	Bruxelles			320	9,736.07
	Belgium	Bruxelles	Anne Dodsworth	6	99.19
	Belgium	Bruxelles	Anne Dodsworth	10	184.00
	Belgium	Bruxelles	Anne Dodsworth	20	680.00
	Belgium	Bruxelles	Anne Dodsworth	40	340.00
	Belgium	Bruxelles	Janet Leverling	6	108.00
	Belgium	Bruxelles	Janet Leverling	6	187.38
	Belgium	Bruxelles	Margaret Peacock	12	126.00

This is correct, but it is also a waste of screen real estate. Eliminating these duplicate values is easy; all you have to do is set the **Width** of the columns that are being grouped on to zero. When you do that, however, you should remember to set the grid's [AllowMerging](#) property to **Nodes**, so the text assigned to the node rows will spill into the visible columns. (Another option would be to assign the node text to the first visible column, but merging is usually a better solution because it allows you to use longer text for the node rows).

Here is the revised code and the final result:

#### To write code in C#

```
C#
void _btnTreeCountryCity_Click(object sender, EventArgs e)
{
    using (new DeferRefresh(_flex))
    {
        // restore original sort (by Country, City, SalesPerson)
        ResetBinding();

        // group by Country, City
        GroupBy("Country", 0); // group by country (level 0)
        GroupBy("City", 1);   // group by city (level 1)

        // hide columns that we grouped on
        // (they only have duplicate values which already appear on the tree nodes)
        // (but don't make them invisible, that would also hide the node text)
        _flex.Cols["Country"].Width = 0;
        _flex.Cols["City"].Width = 0;

        // allow node content to spill onto next cell
        _flex.AllowMerging = AllowMergingEnum.Nodes;

        // add totals per Country, City
        AddTotals(0, "ExtendedPrice"); // extended price per country (level 0)
        AddTotals(0, "Quantity");      // quantity per country (level 0)
        AddTotals(1, "ExtendedPrice"); // extended price per city (level 1)
    }
}
```

```

        AddTotals(1, "Quantity");           // quantity per city (level 1)

        // show outline tree
        _flex.Tree.Column = 0;
        _flex.AutoSizeCol(_flex.Tree.Column);
        _flex.Tree.Show(1);
    }
}

```

	SalesPerson	Quantity	ExtendedPrice
[-]	<b>Argentina</b>	<b>339</b>	<b>8,119.10</b>
[+]	Buenos Aires	339	8,119.10
[-]	<b>Austria</b>	<b>5167</b>	<b>128,003.83</b>
[+]	Graz	4543	104,874.97
[+]	Salzburg	624	23,128.86
[-]	<b>Belgium</b>	<b>1392</b>	<b>33,824.85</b>
[-]	Bruxelles	320	9,736.07
	Anne Dodsworth	6	99.19
	Anne Dodsworth	10	184.00
	Anne Dodsworth	20	680.00
	Anne Dodsworth	40	340.00
	Janet Leverling	6	108.00
	Janet Leverling	6	187.38

The **Country** and **City** columns are now invisible, but their values still appear in the node rows. Collapsing the tree shows totals for each country and city.

## Using the Subtotal Method

We mentioned earlier that you could also create trees using the **C1FlexGrid's Subtotal** method. **TheSubtotal** method performs the same tasks as the **GroupBy** and **AddSubtotals** methods described above, except it does both things in a single step and is therefore a little more efficient.

The code below shows how you can use the **Subtotal** method to accomplish the same thing we did before, only a little faster and without using any helper methods:

### To write code in C#

```

C#
void _btnTreeCountryCity_Click(object sender, EventArgs e)
{
    using (new DeferRefresh(_flex))
    {
        // restore original sort (by Country, City, SalesPerson)
        ResetBinding();

        // group and total by country and city
        _flex.Subtotal(AggregateEnum.Sum, 0, "Country", "ExtendedPrice");
        _flex.Subtotal(AggregateEnum.Sum, 0, "Country", "Quantity");
    }
}

```

```
_flex.Subtotal(AggregateEnum.Sum, 1, "City", "ExtendedPrice");
_flex.Subtotal(AggregateEnum.Sum, 1, "City", "Quantity");

// hide columns that we grouped on
// (they only have duplicate values which already appear on the tree nodes)
// (but don't make them invisible, that would also hide the node text)
_flex.Cols["Country"].Width = 0;
_flex.Cols["City"].Width = 0;
_flex.AllowMerging = AllowMergingEnum.Nodes;

// show outline tree
_flex.Tree.Column = 0;
_flex.AutoSizeCol(_flex.Tree.Column);
_flex.Tree.Show(1);
}
}
```

The **Subtotal** method is very convenient and flexible. It has a number of overloads that allow you to specify which columns should be grouped on and totaled on by index or by name, whether to include a caption in the node rows that it inserts, how to perform the grouping, and so on. The summary below describes the overloads available:

#### 1. **Subtotal(AggregateEnum aggType)**

This version of the method takes only one aggregate type as a parameter. It is useful only for removing existing subtotals before inserting new ones. In this case, the `aggType` parameter is set to `AggregateEnum.Clear`.

#### 2. **Subtotal(AggregateEnum aggType, int groupBy, int totalOn)**

##### **Subtotal(AggregateEnum aggType, string groupBy, string totalOn)**

These are the most commonly used overloads. The parameters are the type of aggregate to insert and the columns to group by and total on. The columns may be referenced by index or by name. The latter is the one we used in the example above.

#### 3. **Subtotal(AggregateEnum aggType, int groupBy, int totalOn, string caption)**

##### **Subtotal(AggregateEnum aggType, string groupBy, string totalOn, string caption)**

These overloads add an extra **caption** parameter. The **caption** parameter determines the text that is added to the new node rows to identify the value being grouped on. By default, the value being grouped on is shown, so if you are grouping by country, the node rows would show "Argentina", "Brazil", and so on. If you set the **caption** parameter to a string such as "Country: {0}", then the node rows would show "Country: Argentina" instead.

#### 4. **Subtotal(AggregateEnum aggType, int groupFrom, int groupTo, int totalOn, string caption)**

##### **Subtotal(AggregateEnum aggType, string groupFrom, string groupTo, string totalOn, string caption)**

These overloads separate the **groupBy** parameter into two: **groupFrom** and **groupTo**. By default, the **Subtotal** method inserts a node row whenever the value of the **groupBy** or any previous column changes.

For example, if you a row has the same value in the "City" column as the previous row, but a different value in the "Country" column, then the **Subtotal** method assumes the rows should be in different groups and inserts a new node row even though the value of the **groupBy** column is the same. These aggregates let you override that behavior and specify the range of columns that should be considered when identifying a group.

## Outline Maintenance

So far we have discussed how to create outlines with trees and totals using the high-level [Subtotal](#) method as well as lower-level **Rows.InsertNode** and **Aggregate** methods.

At this point, it is important to remember that the outline tree is created based on the data, but is not bound to it in any way, and is not automatically maintained when there are changes to the grid or to the data.

If the user modifies a value in the "ExtendedPrice" column, for example, the subtotals will not be automatically updated. If the user sorts the grid, the data will be refreshed and the subtotals will disappear.

There are two common ways to maintain the outlines:

1. Prevent the user from making any changes that would invalidate the outline. This is the easiest option. You would set the grid's [AllowEditing](#), [AllowDragging](#), and [AllowSorting](#) properties to false and prevent any changes that would affect the outline.
2. Update the outline when there are changes to the data or to the grid. You would attach handlers to the grid's [AfterDataRefresh](#), [AfterSort](#), and [AfterEdit](#) events and re-generate the outline appropriately.

Option 2 is usually more interesting since it provides a quick and simple tool for dynamic data analysis. This approach is illustrated by the **Analyze** sample provided with the [C1FlexGrid](#). The sample creates an initial outline and allows users to reorder the columns. When the column order changes, the sample automatically re-sorts the data and re-creates the outline. The user can easily create simple reports showing sales by country, by product, by salesperson, and so on.

## Using the Node class

The [Node](#) class provides a number of methods and properties that can be used to create and manage outline trees. Many of these methods and properties are based on the standard **TreeView** object model, so they should be familiar to most developers.

To obtain a Node object, you can either:

Use the return value of the **Rows.InsertNode** method:

```
var node = _flex.Rows.InsertNode(index, level);
```

Or you can retrieve the node for an existing row using the row's Node property:

```
var node = _flex.Rows[index].IsNode
    ? _flex.Rows[index].Node
    : null;
```

Either way, once you have a Node object you can manipulate it using the following properties and methods:

- **Level:** Gets or sets the node level in the outline tree.
- **Data:** Gets or sets the value in the cell defined by **Node.Row** and the **Tree.Column**.
- **Image:** Gets or sets the image in the cell defined by **Node.Row** and the **Tree.Column**.
- **Checked:** Gets or sets the check state of the cell defined by **Node.Row** and the **Tree.Column**.
- **Collapsed/Expanded:** Gets or sets the node's collapsed/expanded state.

You can also explore the outline structure using the following methods:

- **GetCellRange():** Gets a **CellRange** object that described the range of rows that belong to this node.
- **Children:** Gets the number of child nodes under this node.
- **Nodes:** Gets a node array containing this node's child nodes.

- **GetNode:** Gets the node that has a given relationship to this node (parent, first child, next sibling, and so on).

The discussion above focused on bound scenarios, where the grid is attached to a data source that provides the data. You can also create trees and outlines in unbound scenarios. Things are actually somewhat simpler in this case, since you can turn any row into a node row by setting its **IsNode** property to true.

If the grid is unbound, it owns all the data that is displayed, and you do things that are not possible when a data source owns the data. For example, you can move nodes around the tree using the [Move](#) method as shown by the **TreeNode** sample provided with the [C1FlexGrid](#).

Using nodes in an unbound grid is very similar to using nodes in a regular **TreeView** control.

## Saving, Loading, and Printing

The [C1FlexGrid](#) control has methods that allow you to save, load, and print grids.

## Saving and Loading Grids to Text Files

The [SaveGrid](#) method saves the grid contents to a text file. The method has parameters that control the type of delimiter to use (for example, commas, tabs, custom delimiters), whether to save the fixed cells or only the scrollable cells, and the type of encoding to used (for example, ASCII or Unicode). The resulting text files can later be loaded back into the control, or into other applications that support comma or tab-delimited files (for example, Microsoft Excel).

The [LoadGrid](#) method loads data from text files. You can load text files created with the [SaveGrid](#) method or with other applications.

The format of the text files is fairly simple. Cell contents are saved as formatted strings (exactly as they are displayed on the screen). If the cell text contains quote characters or cell separator characters, the cell is enclosed in quotes. Any quote characters contained in the cell text are doubled. This is also the convention used in Microsoft Excel text files.

Text files do not contain pictures or formatting information.

The [SaveGrid](#) method has a *flags* parameter that allows you to specify whether you want to save the entire grid or only certain parts (scrollable, visible, or selected).

## Saving and Loading Microsoft Excel Files

Starting with version 2.5, you can use the [SaveGrid](#) and [LoadGrid](#) methods to save and load Microsoft Excel files (.xls) as well as text files. This allows you to save formatting information in addition to the data.

To save and load Excel files using the [SaveGrid](#) and [LoadGrid](#) methods, simply set the *format* parameter to **FileFormatEnum.Excel** and call the methods as usual. You don't need to have Microsoft Excel installed on your computer.

Excel files contain "workbooks", which are made up of "worksheets". The [SaveGrid](#) and [LoadGrid](#) methods always save books with a single sheet, and load the first sheet from existing books. If you want additional control over which sheets to load or save, use the [SaveExcel](#), [LoadExcel](#), and [LoadExcelSheetNames](#) methods instead. The process of saving and loading Excel files will convert most data types and formatting information, including row and column dimensions, fonts, colors, formats, and cell alignment. However, not all formatting elements can be converted. For example, the grid will load the values in Excel cells, but it will not load the underlying formulas. Other features such as frozen and merged cells, images, data maps, and cell borders are not translated either.



**Note:** If you use any of the extension methods (such as [SaveExcel](#), [SaveGrid](#), [LoadExcel](#), [LoadGrid](#)), [FileFlags](#) and [FileFlagEnum](#) enumerations, then add reference to [C1.Win.C1FlexGrid.ImportExport.dll.4](#).



## Loading Grids from Databases

You can also load grid data from a database. This is different from data binding, which keeps a live connection between one or more controls and the underlying data source. To load data from a database, you can use **DataReader** objects, as shown below:

### To write code in Visual Basic

#### Visual Basic

```
Private Sub _btnData_Click(ByVal sender As Object, ByVal e As System.EventArgs)
    Handles _btnData.Click

    ' Prepare DataReader.
    Dim strConn As String = "data source=MYMACHINE;initial catalog=Northwind;"
    Dim myConn As New SqlClient.SqlConnection(strConn)
    Dim myCMD As New SqlClient.SqlCommand("SELECT * FROM Employees", myConn)
    myConn.Open()
    Dim myReader As SqlClient.SqlDataReader = myCMD.ExecuteReader()

    ' Build the grid structure from the DB schema.
    Dim dt As DataTable = myReader.GetSchemaTable()
    _flex.Cols.Count = 1
    Dim dr As DataRow
    For Each dr In dt.Rows
        Dim c As Cl.Win.C1FlexGrid.Column = _flex.Cols.Add()
        c.Caption = (c.Name <= CStr(dr("ColumnName")))
        c.DataType = CType(dr("DataType"), Type)
    Next dr

    ' Populate the grid.
    _flex.Rows.Count = 1
    Dim row As Integer = 1
    Dim cols As Integer = dt.Columns.Count
    Dim v As Object() = CType(Array.CreateInstance(GetType(Object), cols), Object())
    While myReader.Read()
        myReader.GetValues(v)
        _flex.AddItem(v, row + 1, 1)
    End While

    ' Cleanup.
    _flex.AutoSizeCols()
    myReader.Close()
    myConn.Close()
End Sub
```

### To write code in C#

#### C#

```
private void _btnData_Click(object sender, System.EventArgs e)
{
```

```
// Prepare DataReader.
string strConn = "data source=MYMACHINE;initial catalog=Northwind;";
System.Data.SqlClient.SqlConnection myConn = new
System.Data.SqlClient.SqlConnection(strConn);
System.Data.SqlClient.SqlCommand myCMD = new
System.Data.SqlClient.SqlCommand("SELECT * FROM Employees", myConn);
myConn.Open();
System.Data.SqlClient.SqlDataReader myReader = myCMD.ExecuteReader();

// Build the grid structure from the DB schema.
DataTable dt = myReader.GetSchemaTable();
_flex.Cols.Count = 1;
foreach (DataRow dr in dt.Rows)
{
    Column c = _flex.Cols.Add();
    c.Caption = c.Name = (string)dr["ColumnName"];
    c.DataType = (Type)dr["DataType"];
}

// Populate the grid.
_flex.Rows.Count = 1;
int row = 1;
int cols = dt.Columns.Count;
object[] v = (object[])Array.CreateInstance(typeof(object), cols);
while (myReader.Read())
{
    myReader.GetValues(v);
    _flex.AddItem(v, row++, 1);
}

// Cleanup.
_flex.AutoSizeCols();
myReader.Close();
myConn.Close();
}
```

## Printing Grids

Use the [PrintGrid](#) method to print the contents of the grid. The method has parameters that allow you to select the scaling mode, whether to display print/preview dialog boxes, set headers and footers, and so on.

The [PrintParameters](#) property exposes additional printing properties such as the font to use for headers and footers, and a .NET Framework **PrintDocument** object that can be used to select the printer, paper size and orientation, margins, and so on.

The code below uses the [PrintParameters](#) property to set up the page orientation, margins, header and footer fonts. Then it calls the [PrintGrid](#) method to display a print preview dialog window:

### To write code in Visual Basic

Visual Basic

```
' Get the grid's PrintDocument object.
```

```

Dim pd As Printing.PrintDocument
pd = _flex.PrintParameters.PrintDocument()

' Set up the page (landscape, 1.5" left margin).
With pd.DefaultPageSettings
    .Landscape = True
    .Margins.Left = 150
End With

' Set up the header and footer fonts.
_flex.PrintParameters.HeaderFont = New Font("Arial Black", 14, FontStyle.Bold)
_flex.PrintParameters.FooterFont = New Font("Arial Narrow", 8, FontStyle.Italic)

' Preview the grid.
_flex.PrintGrid("C1FlexGrid", C1.Win.C1FlexGrid.PrintGridFlags.FitToPageWidth +
C1.Win.C1FlexGrid.PrintGridFlags.ShowPreviewDialog, "C1FlexGrid" + Chr(9) + Chr(9) +
Format(DateTime.Now, "d"), Chr(9) + Chr(9) + "Page {0} of {1}")

```

### To write code in C#

```

C#
// Get the grid's PrintDocument object.
System.Drawing.Printing.PrintDocument pd = _flex.PrintParameters.PrintDocument;

// Set up the page (landscape, 1.5" left margin).
pd.DefaultPageSettings.Landscape = true;
pd.DefaultPageSettings.Margins.Left = 150;

// Set up the header and footer fonts.
_flex.PrintParameters.HeaderFont = new Font("Arial Black", 14, FontStyle.Bold);
_flex.PrintParameters.FooterFont = new Font("Arial Narrow", 8, FontStyle.Italic);

// Preview the grid.
_flex.PrintGrid("C1FlexGrid", PrintGridFlags.FitToPageWidth |
PrintGridFlags.ShowPreviewDialog, "C1FlexGrid\t\t" +
Microsoft.VisualBasic.Strings.Format(DateTime.Now, "d"), "\t\tPage {0} of {1}");

```

## C1FlexGrid Filtering

Data filtering in grids is usually of two types:

- **Header-Based Filter:** A filter icon appears over each column that has a filter applied to it. Users may see and edit the filters by clicking the filter icon. This is the mechanism used by Windows 7/Vista and by the [C1FlexGrid](#) control. The main benefits of this type of filter are: (1) the user can see which columns are being filtered, (2) the filtering does not require extra real estate on the screen, and (3) this type of filter allows for better filter editors and easier customization.
- **Filter Row:** A filter row remains always visible and allows users to type values or expressions directly into the row. The main benefit of this type of filter is that users can always see which columns are being filtered and what the current filter criteria are. The main disadvantage is that the filter takes up some real estate and may interfere with the regular grid operation. Although filter rows are not built into the [C1FlexGrid](#) control,

implementing them is relatively easy. We provide a **FilterRow** sample that shows how to do it.

The code samples presented below were mostly taken from two new samples included with the product: **ColumnFilters** and **CustomFilters**. Please refer to the samples for complete projects that show the features in action.

## AllowFiltering property

To implement the header-based filters, the **C1FlexGrid** control follows the same pattern used to implement column moving and sizing. The grid has a new **AllowFiltering** property that controls filtering at a control level, and the grid's **Column** object also has an **AllowFiltering** property that controls filtering at the column level.

To enable simple filtering scenarios, users simply set the grid's **AllowFiltering** property to **True**. They can then disable or customize the filtering behavior for specific columns by changing the value of the column's **AllowFiltering** property. The column's **AllowFiltering** property may be set to one of the following values:

- **Default:** The grid automatically creates a filter of type **ColumnFilter**. This filter combines a **ValueFilter** and a **ConditionFilter**, both described below.
- **ByValue:** The grid automatically creates a filter of type **ValueFilter**. This filter contains a list of values that should be displayed. Any values not present on the list are hidden from the end user.
- **ByCondition:** The grid automatically creates a filter of type **ConditionFilter**. This filter specifies two conditions such as "greater than" or "contains". The conditions may be combined with an AND or an OR operator.
- **Custom:** The grid does not create a filter automatically. The developer is expected to instantiate a filter and explicitly assign it to the column's **Filter** property.
- **None:** The column cannot be filtered.

By default, the **C1FlexGrid** control localizes the column filter editor to use the language specified by the **CurrentUICulture** setting. However, you can use the **Language** property to override the default and specify the language that should be used when the grid displays the column filter editor.

## Value Filter

The **ValueFilter** is conceptually very simple. It contains a list of values, and only the values listed are displayed on the grid. If the list is set to null, the filter is de-activated and all values are displayed. This type of filter is recommended for filtering columns that contain discrete values such as names or enumerations.

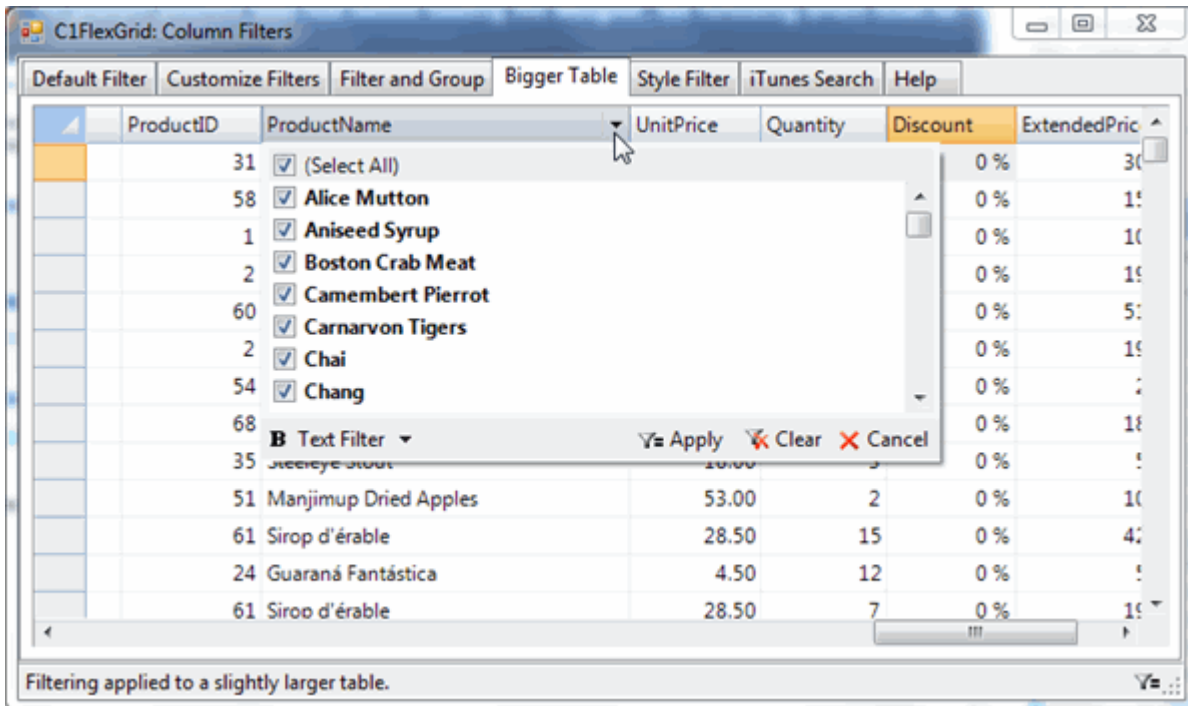
The **ValueFilter** editor consists of a list of values with checkboxes. Users may check or uncheck all values at once. Navigating long lists is easy with the advanced built-in keyboard navigation. The editor implements a flexible search buffer that allows users to locate values by typing any part of the value. For example, typing "Hilton" will select the next value that contains "Hilton", including "New York Hilton", "Prince Edward Hilton", or "Paris Hilton". Also, typing **ctrl+Up** or **ctrl+Down** will navigate to the next or previous checked item.

Values are displayed on the list using the **Format** currently assigned to the column.



**Note:** When you set filters on more than one column of the grid, the **ValueFilter** editor only displays the values available for filtering after the previous filters are set.

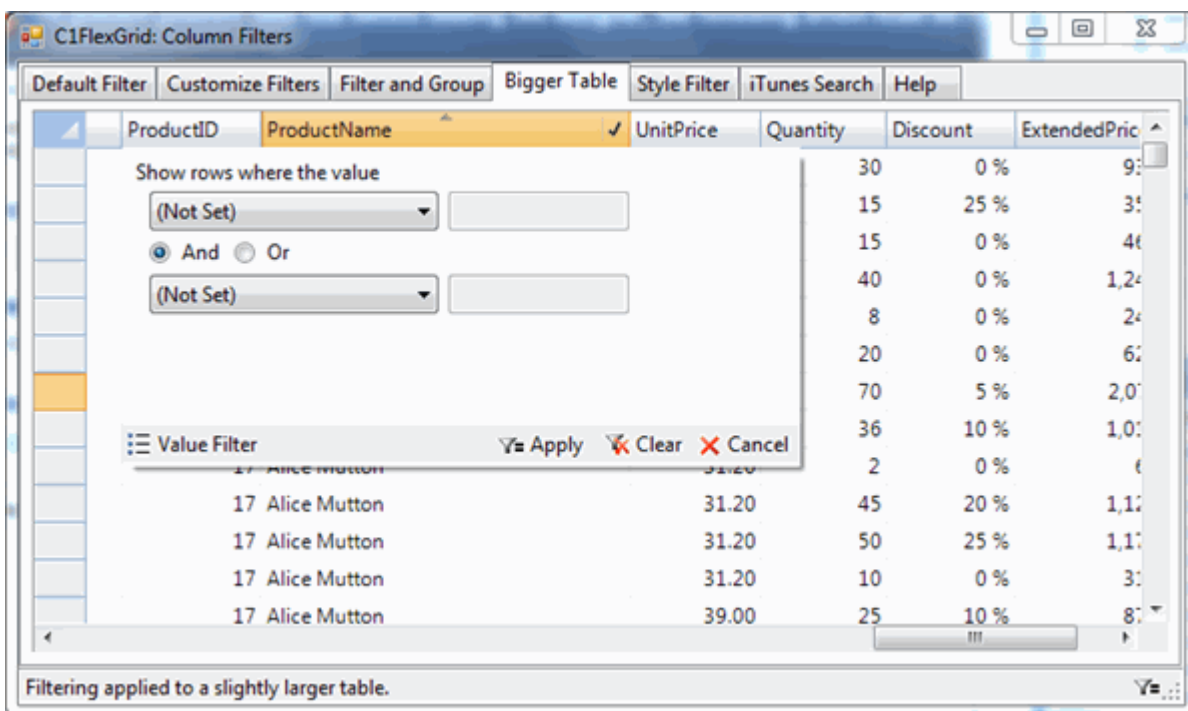
This image shows the **ValueFilter** editor.



## Condition Filter

The ConditionFilter is more flexible. Instead of selecting specific values, it allows users to specify up to two conditions using operators such as "greater than", "starts with", or "contains". This type of filter is recommended for filtering columns that contain "continuous" values such as numeric or date/time values.

This image shows the **ConditionFilter** editor.



Columns that have filters applied display the filter icon on their headers even when the mouse is not over them. You can see this in the images, where the **ProductName** and **Quantity** column headers show the filter icon.

The built-in filters support automatic localization in the following languages: English, Spanish, French, Italian, Portuguese, German, Dutch, Russian, Japanese, Greek, Danish, Finnish, Norwegian, Swedish, Arabic, Polish, Chinese (*Traditional*: Taiwan, Hong Kong, Macao; *Simplified*: PRC, Singapore), Turkish, Persian (Farsi), Korean, and Hebrew. The localization is built-in and does not require satellite DLLs.

## Custom Filters

Custom filters may be created to handle specialized values. For example, custom filters would be recommended to filter on colors, geographic, or custom data types.

To create a custom filter, the developer must create two classes:

- **Filter**: This class must implement the [IC1ColumnFilter](#) interface, which specifies methods used to apply the filter to a specific value, to reset the filter, and to return an editor to be used for viewing and editing the filter's parameters.
- **Filter Editor**: This class must inherit from **Control** and must implement the [IC1ColumnFilterEditor](#) interface, which specifies methods used to initialize the editor and to apply changes to the filter.

The **CustomFilters** sample contains implementations of three custom filters used for filtering values of type **Color**, **DateTime**, and **string**.

## Managing filters programmatically

As we mentioned earlier in this document, setting the grid's [AllowFiltering](#) property to **True** is enough to enable column filtering on all columns. In many cases however, you may want finer control over filtering. This can be achieved by modifying the [AllowFiltering](#) and [Filter](#) properties on individual columns.

For example, the code below enables filtering but restricts the filtering to columns of type **string**:

### To write code in C#

```
C#  
  
// bind and configure grid  
_flex.DataSource = dtProducts;  
_flex.Cols["UnitPrice"].Format = "#,###.00";  
  
// enable filtering  
_flex.AllowFiltering = true;  
  
// restrict filtering to columns of type 'string'  
foreach (Column c in _flex.Cols)  
{  
    c.AllowFiltering = c.DataType == typeof(string)  
        ? AllowFiltering.Default  
        : AllowFiltering.None;  
}
```

You can customize the filtering process further by creating filters and assigning them to columns, or by retrieving existing filters and modifying their properties. For example, the code below creates a [ConditionFilter](#), configures it to select all items that start with the letter "C", and then assigns the new filter to the "ProductName" column:

### To write code in C#

```
C#
```

```
// create a new ConditionFilter
var filter = new ConditionFilter();

// configure filter to select items that start with "C"
filter.Condition1.Operator = ConditionOperator.BeginsWith;
filter.Condition1.Parameter = "C";

// assign new filter to column "ProductName"
_flexCustom.Cols["ProductName"].Filter = filter;
```

## Applying filters programmatically

Filters are applied when the user edits them, and also when they are applied to a column. They are not automatically applied when the data changes. To apply the filters to the data currently loaded on the grid, call the grid's [ApplyFilters](#) method.

For example, the code below enables an **ApplyFilter** button when the user edits the data on the grid. Clicking the **ApplyFilter** button applies the filter and disables the button until the next change.

### To write code in C#

```
C#

public Form1()
{
    InitializeComponent();

    // get some data
    var da = new OleDbDataAdapter("select * from products", GetConnectionString());
    var dtProducts = new DataTable();
    da.Fill(dtProducts);

    // bind the grid to the data
    _flex.DataSource = dtProducts;

    // enable filtering
    _flex.AllowFiltering = true;

    // monitor changes to enable ApplyFilter button
    _flex.AfterEdit += _flex_AfterEdit;
}
```

The code above binds the grid to a data source, enables filtering by setting the [AllowFiltering](#) property to **True**, and connects an event handler to the **AfterEdit** event. The event handler implementation follows:

### To write code in C#

```
C#

void _flex_AfterEdit(object sender, Cl.Win.ClFlexGrid.RowColEventArgs e)
{
    foreach (Cl.Win.ClFlexGrid.Column c in _flex.Cols)
    {
        if (c.ActiveFilter != null)
```

```
        {
            _btnApplyFilters.Enabled = true;
            break;
        }
    }
}
```

This code scans all columns to determine whether a filter is defined for any column. If an active filter is detected, the code enables the button that applies the filter to the current data. When the button is clicked, the following event handler executes:

#### To write code in C#

C#

```
private void _btnApplyFilters_Click(object sender, EventArgs e)
{
    _flex.ApplyFilters();
    _btnApplyFilters.Enabled = false;
}
```

The code simply applies all the active filters and disables the button until the next change.

If instead of requiring a button click you simply wanted to apply the filter after every edit, you could call the **ApplyFilter** directly from the **AfterEdit** event handler, as shown below:

#### To write code in C#

C#

```
void _flex_AfterEdit(object sender, C1.Win.C1FlexGrid.RowColEventArgs e)
{
    _flex.ApplyFilters();
}
```

## Customizing filter behavior

When filters are applied, the grid hides rows by setting their **Visible** property to **False**. But the grid also fires **BeforeFilter** and **AfterFilter** events that allow you to customize the filtering behavior. For example, assume that instead of showing and hiding rows you wanted to apply different styles to indicate whether the rows pass the filter or not. This can be accomplished easily using this code:

#### To write code in C#

C#

```
public Form1()
{
    InitializeComponent();

    // configure grid
    _flex.AllowFiltering = true;
    _flex.DataSource = dtInvoices;

    // create style for rows excluded by the filter
```



```
var cs = _flexStyles.Styles.Add("filteredOut");
cs.BackColor = Color.LightSalmon;
cs.ForeColor = Color.DarkOrange;

// connect handlers for the before and after filter events
_flex.BeforeFilter += _flex_BeforeFilter;
_flex.AfterFilter += _flex_AfterFilter;
}
```

The code creates a custom style that will be used to show rows that did not pass the filter (instead of making them invisible). Next, the code attaches handlers for the BeforeFilter and AfterFilter events. The event handlers are listed below:

#### To write code in C#

C#

```
// suspend painting before filter is applied
void _flex_BeforeFilter(object sender, CancelEventArgs e)
{
    _flexStyles.BeginUpdate();
}

// apply styles after the filter has been applied
void _flexStyles_AfterFilter(object sender, EventArgs e)
{
    // get style used to show filtered out rows
    var cs = _flex.Styles["filteredOut"];

    // apply style to all rows
    for (int r = _flexStyles.Rows.Fixed; r < _flexStyles.Rows.Count; r++)
    {
        var row = _flexStyles.Rows[r];
        if (row.Visible)
        {
            // normal row, reset style
            row.Style = null;
        }
        else
        {
            // filtered row, make visible and apply style
            row.Visible = true;
            row.Style = cs;
        }
    }

    // resume updates
    _flexStyles.EndUpdate();
}
```

The BeforeFilter event handler calls the new BeginUpdate method to prevent the grid from repainting itself until we are done applying the custom style to the rows that are filtered out. The BeginUpdate and EndUpdate methods replace the Redraw property which has been deprecated.

The AfterFilter event handler starts by retrieving the style we created to show the rows that have been filtered out. It then scans the rows on the grid and applies the new style to all rows that have the Visible property set to **False**. These are the rows that were hidden by the filter. Once this is done, the code calls EndUpdate to restore grid updating.

## Customizing the Filtering UI

We believe the default filtering behavior and UI will address the vast majority of scenarios that involve column filtering. However, you can use the column filter classes independently, implementing your own custom user interface.

For example, the code below shows how you can use the [ConditionFilter](#) class to implement an iTunes-style search box for the [C1FlexGrid](#). This type of search allows users to type in a value and automatically filters the grid rows to show the rows that contain the search string in any column.

To implement the iTunes-style search, we start with a text box that contains the text that will be used as a filter parameter. We also define a timer that will apply the filter a few milliseconds after the users stops typing into the text box:

### To write code in C#

```
C#
public Form1()
{
    InitializeComponent();

    // configure timer to apply the filter 1/2 second after the
    // user stops typing into the search box
    _timer.Interval = 500;
    _timer.Tick += t_Tick;

    // monitor changes to the search box
    _txtSearch.TextChanged += _txtSearch_TextChanged;
}

// re-start timer when the text in the search box changes
void _txtSearch_TextChanged(object sender, EventArgs e)
{
    _timer.Stop();
    _timer.Start();
}
```

Now that the timer is configured, all we need to do is create and apply the filter when the timer ticks:

### To write code in C#

```
C#
// apply filter when the timer ticks
void t_Tick(object sender, EventArgs e)
{
    // done for now...
    _timer.Stop();

    // configure filter
    var filter = new C1.Win.C1FlexGrid.ConditionFilter();
}
```

```
filter.Condition1.Operator = C1.Win.C1FlexGrid.ConditionOperator.Contains;
filter.Condition1.Parameter = _txtSearch.Text;

// apply filter
_flex.BeginUpdate();
for (int r = _flex.Rows.Fixed; r < _flex.Rows.Count; r++)
{
    bool visible = false;
    for (int c = _flex.Cols.Fixed; c < _flex.Cols.Count; c++)
    {
        if (filter.Apply(_flex[r, c]))
        {
            visible = true;
            break;
        }
    }
    _flex.Rows[r].Visible = visible;
}
_flex.EndUpdate();
}
```

## C1FlexGrid Property Groups

The [C1FlexGrid](#) control has a rich set of properties, methods, and events. You do not have to know all of them in order to use the control effectively.

The reference below shows the most important properties, methods, and events grouped by usage type. Some elements appear in more than one group. Select specific elements below for more details.

### 1. Grid Layout

[Rows](#), [Cols](#), [AutoSizeCols](#), [ScrollBars](#)

### 2. Cursor and Selection

[SelectionMode](#), [Select](#), [ShowCell](#), [Row](#), [Col](#), [RowSel](#), [ColSel](#), [MouseRow](#), [MouseCol](#), [BeforeRowColChange](#), [AfterRowColChange](#), [BeforeSelChange](#), [AfterSelChange](#), [KeyActionTab](#), [KeyActionEnter](#)

### 3. Editing

[AllowEditing](#), [ComboList](#), [EditMask](#), [BeforeEdit](#), [StartEdit](#), [ValidateEdit](#), [AfterEdit](#), [StartEditing](#), [FinishEditing](#), [Editor](#), [CellButtonClick](#), [KeyDownEdit](#), [KeyPressEdit](#), [KeyUpEdit](#), [ChangeEdit](#)

### 4. Getting and Setting Values

[Item](#) (indexer), [GetData](#), [GetDataDisplay](#), [SetData](#), [GetCellRange](#), [GetCellImage](#), [SetCellImage](#), [Clip](#), [FindRow](#), [Aggregate](#), [CellChanged](#)

### 5. User Interface

[AllowEditing](#), [AllowMerging](#), [AllowResizing](#), [AllowDragging](#), [AllowSorting](#), [BeforeSort](#), [AfterSort](#), [AutoSearch](#), [AutoSearchDelay](#), [BeforeDragColumn](#), [AfterDragColumn](#), [BeforeDragRow](#), [AfterDragRow](#), [BeforeResizeColumn](#), [AfterResizeColumn](#), [BeforeResizeRow](#), [AfterResizeRow](#), [ShowScrollTip](#)

### 6. Outlining and Summarizing

[Subtotal](#), [Tree](#), [IsNode](#), [Level](#), [Collapsed](#), [BeforeCollapse](#), [AfterCollapse](#)

## 7. **Merging Cells**

[AllowMerging](#)

## 8. **Data Binding**

[DataSource](#), [DataMember](#), [AfterDataRefresh](#), [AutoResize](#), [GridError](#)

## 9. **Saving, Loading, and Printing Grids**

[LoadGrid](#), [SaveGrid](#), [LoadExcel](#), [SaveExcel](#), [ClipSeparators](#), [PrintGrid](#)

## 10. **OLE Drag Drop**

[DragMode](#), [DropMode](#), [BeforeMouseDown](#)

## 11. **Filtering**

## Data Binding

Data binding is a process that allows one or more data consumers to be connected to a data provider in a synchronized manner. If you move the cursor on a data-bound grid, other controls connected to the same data source will change to reflect the new current record. If you edit a value on a data-bound grid, other controls connected to the same data source will change to reflect the new value.

**C1FlexGrid** supports data binding to ADO.NET data source objects such as **DataTable**, **DataView**, **DataSet**, and **DataViewManager**.

**C1FlexGrid** also supports data binding to **DataObjects for WinForms** components such as **C1ExpressTable**, **C1ExpressView**, **C1ExpressConnection**, **C1DataView**, **C1DataTableSource** and **C1DataSet**.

To bind the grid to a data source, assign the data source object to the grid's **DataSource** property. If the data source object contains more than one table, you must also set the **DataMember** property a string that specifies which table should be used.

Alternatively, you can assign both properties simultaneously with a single call to the **SetDataBinding** method.

When you assign a new data source to the grid, it will automatically refresh its columns to bind to the columns available in the data source. You can then customize the columns by moving, hiding, or deleting them. You can also set column properties such as their **Width**, **EditMask** and **Format**.



For an example of reordering the grid columns after binding to a data source, see the **ColumnOrder** sample on [GrapeCity website](#)

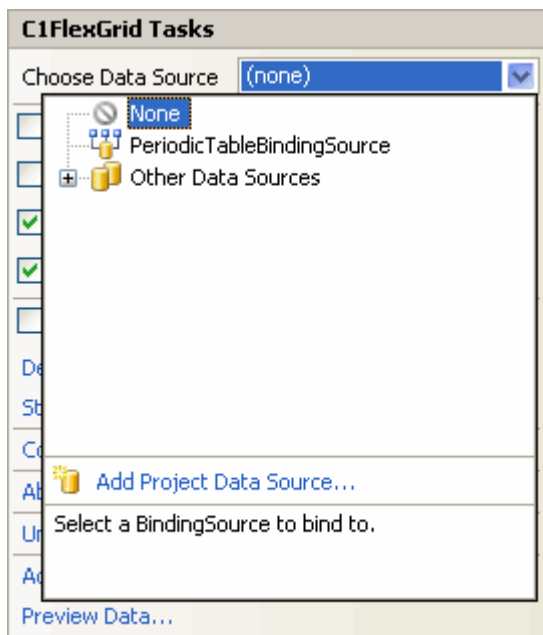
For details about creating ADO.NET data source objects, please refer to the .NET Framework documentation.

For details about using **DataObjects for WinForms**, see the **DataObjects for WinForms** documentation included in **ComponentOne Studio Enterprise** and **ComponentOne Studio WinForms Edition**.

## Binding to a Data Source

Without writing a line of code, you can easily bind **C1FlexGrid** to a data source using the **Data Source Configuration Wizard** in Visual Studio. There are two ways to access the wizard, either through the **DataSource** property in the Properties window or the **Choose Data Source** box in the **C1FlexGrid Tasks** menu. For more information on the **C1FlexGrid Tasks** menu, see [C1FlexGrid Tasks Menu](#).

Clicking the drop-down arrow next to either the **DataSource** property in the Properties window or the **Choose Data Source** box in the **C1FlexGrid Tasks** menu allows you to select from a list of available data sources or to add a data source to your project. To add a data source to your project, click **Add Project Data Source** to open the **Data Source Configuration Wizard**.

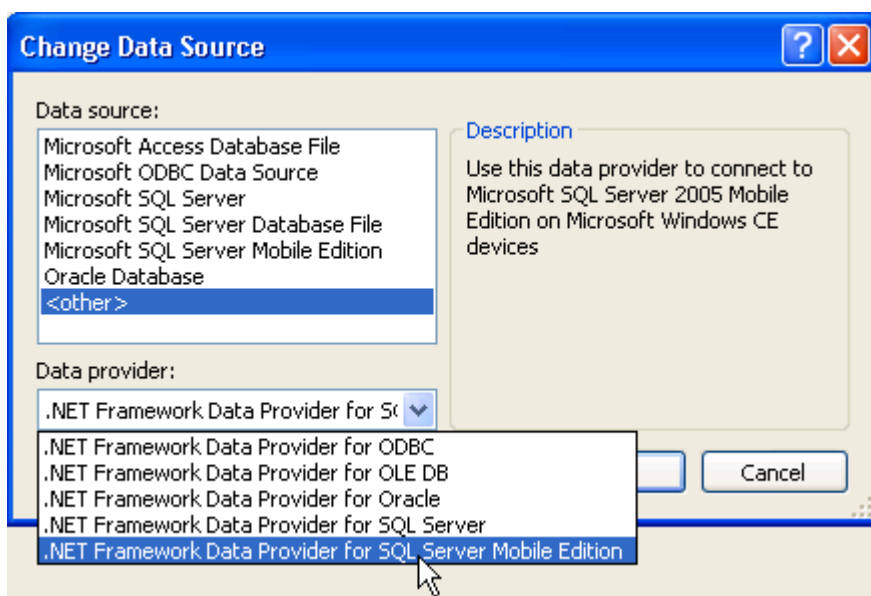


The **Data Source Configuration Wizard** walks you through adding a data source step by step. The **Choose a Data Source Type** page allows you to choose which type of application you would like to gather information from. The default selection is **Database**.

On the **Choose Your Data Connection** page, you can specify the location of the database. If you have not already connected to the database, you can specify a new connection by clicking the **New Connection** button. Clicking the **New Connection** button opens the **Add Connection** dialog box where you can browse to the location of your database and test the connection.

**Mobile device users click here for additional information.**

If you are using the Mobile edition of **FlexGrid for WinForms**, in the **Add Connection** dialog box, change the **Data source** to one that is supported by the Compact Framework, such as the .NET Framework Data Provider for SQL Server Mobile Edition.

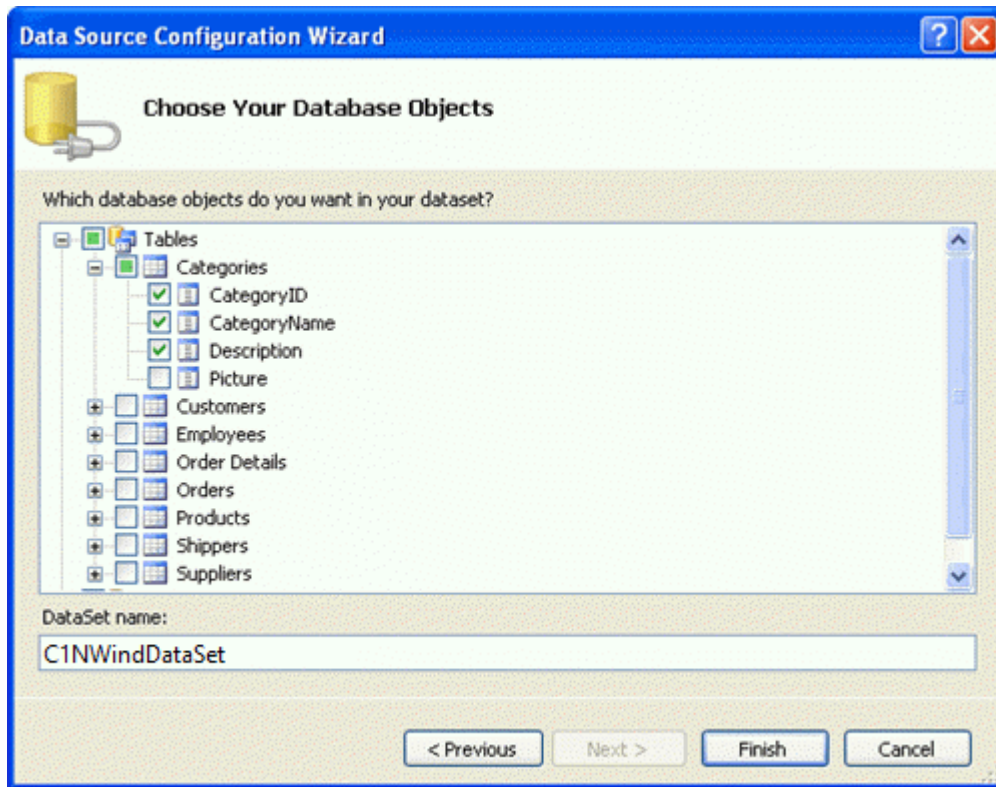


Then you can use a supported database, for example, a SQL Server Mobile Edition Database. Microsoft provides a `Northwind.sdf`, by default, in the `C:\Program Files\Microsoft Visual Studio 8\SmartDevices\SDK\SQL`

Server\Mobile\v3.0 folder for Microsoft Visual Studio 2005.

The wizard saves and names your connection on the **Save the Connection String to the Application Configuration File** page. On this page you can specify a name or use the default provided by the wizard.

On the **Choose Your Database Objects** page you can specify which tables and fields you want to include in your DataSet. You can also provide a name for your DataSet or use the default provided by the wizard.



The wizard creates the DataSet, BindingSource, and TableAdapter, and adds them to your project. Click the drop-down arrow again next to the **DataSource** property or the **Choose Data Source** box in the **C1FlexGrid Tasks** menu and then select the data source. If you double-click the form, you will also notice that the code to fill the database is automatically generated in the **Form\_Load** event.

## Storing and Retrieving Data

The **C1FlexGrid** control can be used in bound or unbound mode. In bound mode, the grid is connected to a data source, and all the data displayed on the grid comes from the data source. In this mode, changing data on the grid changes it in the underlying data source. In unbound mode, the grid manages its own data source.

In either bound or unbound modes, the easiest way to access data in the **C1FlexGrid** is using the **Row** and **Column** indexers. The indexers allow you to specify a cell in a row or column from which to get or set the data stored there. For example, the following code selects the data in the second cell of a row:

### To write code in Visual Basic

Visual Basic

```
Row[2].Selected = True
```

### To write code in C#

C#

```
Row[2].Selected = true;
```

The [Item](#) property is another easy way to access data in the C1FlexGrid. The Item property is an indexer that takes row and column indices and gets or sets the data stored in the cell. (You can also use column names as indices.) For example, the following code stores row numbers in the first grid column:

### To write code in Visual Basic

Visual Basic

```
Dim r As Integer
For r = _flex.Rows.Fixed To _flex.Rows.Count - 1
    _flex(r, 0) = r
Next
```

### To write code in C#

C#

```
for (int r = _flex.Rows.Fixed; r <= _flex.Rows.Count - 1; r++)
{
    _flex[r, 0] = r;
}
```

When you assign a value to a cell, the grid tries to convert that value into the column's specified [DataType](#). If the conversion fails, the grid fires the [GridError](#) event and does not change the cell. You can override this behavior using the [SetData](#) method and setting the *coerce* parameter to **False**.

When you retrieve data using the indexers, the grid returns the actual data stored in the cell. To retrieve a string containing the formatted version of the data (what the grid displays to the user), use the [GetDataDisplay](#) method.

You can also set and retrieve the contents of the selection using the [Clip](#) property. This property is especially useful in handling the clipboard and drag-drop operations. By default, the Clip property returns a string with tab characters (Chr(9)) between cells and return characters (Chr(13)) between rows. To use different delimiters, change the [ClipSeparators](#) property.

Finally, you can set and retrieve the contents of arbitrary cell ranges using [CellRange](#) objects.



## FlexGrid for WinForms Samples

Please be advised that this ComponentOne software tool is accompanied by various sample projects and/or demos which may make use of other development tools included with the ComponentOne Studio.

Please refer to the pre-installed product samples through the following path:

### Documents\ComponentOne Samples\WinForms

Click one of the following links to view a list of **C1FlexGrid** samples:

#### Visual Basic Samples

Sample	Description
AutoSizeEdit	This sample demonstrates resizing the editor while the user types. This sample uses the <a href="#">C1FlexGrid</a> control.
BarChart	This sample uses OwnerDrawn cells to build a chart based on the grid data. This sample uses the C1FlexGrid control.
Blinker	This sample shows how you can use CellStyles to make cells blink. The sample creates a custom style and uses a timer to change the foreground color on that style. This causes all cells that have that style to blink. This sample uses the C1FlexGrid control.
BoundDelete	This sample deletes rows from a bound FlexGrid. This sample uses the C1FlexGrid control.
BoundFinishEdit	This sample demonstrates how to commit changes to a DataRow after editing. This sample uses the C1FlexGrid control.
BoundImage	This sample shows how to bind the grid to image fields stored as OLE objects. This sample uses the C1FlexGrid control.
C1FlexGridPrintable	This sample provides a printable (via C1Preview) C1FlexGrid control. This sample uses the C1FlexGrid control and calls the C1.C1Preview namespace.
CellBorders	This sample uses OwnerDraw to provide custom cell borders. This sample uses the C1FlexGrid control.
CellDataType	This sample assigns cell types (and editors) on a per-cell basis. This sample uses the C1FlexGrid control.
CellMerging	This sample shows various types of cell-merging in a C1FlexGrid control. This sample uses the C1FlexGrid control.
CellNotes	This sample attaches notes to cells on a C1FlexGrid and implements Excel-style cell notes using two classes: <b>CellNote</b> and <b>CellNoteManager</b> .
ColumnEditor	This sample exposes the C1FlexGrid design-time column editor in your controls. This sample uses the C1FlexGrid control.
CustomDataMap	This sample demonstrates how to customize the options in drop-down lists when using DataMaps. This sample uses the C1FlexGrid control.
CustomEditor	This sample shows how to implement your own cell editor. This sample uses the C1FlexGrid control.
CustomMerge	This sample shows how to implement your own merging logic to create a TV-guide display. This sample calls the C1.Win.C1FlexGrid namespace.

Sample	Description
CustomMerge2	This sample shows how to implement your own merging logic to mix merging modes. This sample calls the C1.Win.C1FlexGrid namespace.
CustomMerge3	This sample shows how to customize grouping by overriding the GetData method. This sample calls the C1.Win.C1FlexGrid namespace.
CustomMerge4	This sample shows how to customize grouping by overriding the GetData method. This sample calls the C1.Win.C1FlexGrid namespace.
CustomPrint	This sample uses the CreateImage method to print grids with arbitrary row and column breaks. This sample uses the C1FlexGrid control.
CustomTree	This sample uses the C1FlexGrid control to build a tree. This sample uses the C1FlexGrid control.
DataIndex	This sample uses the Row.DataIndex property to get the underlying data row. This sample uses the C1FlexGrid control.
DataMap	This sample uses data-mapped columns when bound to a data source. This sample uses the C1FlexGrid control.
DataReader	This sample populates a grid from a DataReader. This sample uses the C1FlexGrid control.
DataTable	This sample shows how you can create, populate, and bind a DataTable object to the grid (including add/remove rows). This sample uses the C1FlexGrid control.
DataTree	This sample binds the grid to a hierarchical data source and show details in child grids.
DBDynamicStyles	This sample assigns styles to grid cells based on their contents. This sample uses the C1FlexGrid control.
DBImageField	This sample shows images stored in a DataTable. This sample uses the C1FlexGrid control.
DBSchemaChange	This sample shows how to test grid response to changes in the data source object. This sample uses the C1FlexGrid control.
DBTree	This sample shows how to display bound data in a hierarchical tree view. This sample uses the C1FlexGrid control.
DragDrop	This sample shows how to customize automatic Drag & Drop. This sample uses the C1FlexGrid control.
DragDropManual	This sample shows how to implement manual OleDragDrop between controls and within the grid.
DragRow	This sample shows how to drag rows between grids. This sample uses the C1FlexGrid control.
DynamicStyles	This sample shows how to assign styles to a cell based on the contents of another cell. This sample uses the C1FlexGrid control.
Editing	This sample demonstrates how to edit cells using textboxes, lists, masks, and checkboxes. This sample uses the C1FlexGrid control.
FilterRow_With_Totals	This sample shows a how to add totals to a bound grid with a FilterRow.
FlexByRow	This sample sets up a grid in 'transposed' format. In this sample data types, edit

Sample	Description
	masks, formats, and other properties usually assigned to grid columns are assigned to grid rows. This sample uses the C1FlexGrid control.
FlexGroup	This sample shows how you can implement Outlook-style grouping and filtering using the C1FlexGrid. This sample uses the C1FlexGrid control.
FreezeBottom	This sample demonstrates how to show frozen rows at the bottom of a grid. This sample uses the C1FlexGrid control.
GridChart	This samples show how you can attach a FlexGrid to a C1Chart control. This sample uses the C1FlexGrid and <b>C1Chart</b> controls.
HierData	This sample shows how you can bind to hierarchical data sources (master-detail style). This sample uses the C1FlexGrid control.
Hyperlink	This sample shows how you can add hyperlinks to cells in the C1FlexGrid. This sample uses the C1FlexGrid control.
LoadSpeed	This sample shows three techniques for loading data into a C1FlexGrid.
MapAndGroup	This sample shows grouping and sorting behavior when using data-mapped columns. This sample uses the C1FlexGrid control.
MergeStyles	This sample merges CellStyles assigned to rows, columns, and cells. This sample uses the C1FlexGrid control.
MultiColumnDictionary	This sample uses the MultiColumnDictionary class to implement multi-column drop-downs.
Outline	This sample shows how to hide repeated data in an outline. This sample uses the C1FlexGrid control.
PdfExportVB	This sample uses C1Pdf method to export C1FlexGrids to PDF files.
ProductTree	This sample demonstrates how to build a custom tree with product information. This sample uses the C1FlexGrid control.
RTFGrid	This sample shows how to display RTF text in grid cells. This sample uses the C1FlexGrid control.
ScrollPosition	This sample shows how the ScrollPosition property works. This sample uses the C1FlexGrid control.
SelectionMode	This sample shows the effect of the SelectionMode property. This sample uses the C1FlexGrid control.
Sorting	This sample shows several sorting techniques. This sample uses the C1FlexGrid control.
Splits	This sample demonstrates how to split a C1FlexGrid into multiple views. This sample uses the C1FlexGrid control.
StartEditing	This sample implements a grid that stays in cell-editing mode. This sample uses the C1FlexGrid control.
Styles	This sample shows how you can use Styles to customize the appearance of the C1FlexGrid. This sample uses the C1FlexGrid control.
Subtotals	This sample creates subtotals for multiple columns. This sample uses the C1FlexGrid control.

Sample	Description
Transpose	This sample shows how to transpose data in a grid. This sample uses the C1FlexGrid control.
TreeCheck	This sample shows how to add checkboxes to a grid tree. This sample uses the C1FlexGrid control.
TreeNode	This sample shows how to manage an outline tree using the C1FlexGrid Node objects. This sample uses the C1FlexGrid control.
Validate	This sample shows how to use the ErrorProvider control with the C1FlexGrid. This sample uses the C1FlexGrid control.
ZoomCell	This sample shows how to zoom in on the selected cell.

## C# Samples

Sample	Description
AcceleratorCaption	This sample shows how to add accelerator keys to grid captions.
Analyze	This sample shows how to provide dynamic data sorting and grouping. This sample uses the C1FlexGrid control.
AnimagedGif	This sample shows how to display animated gifs in grid cells.
AutoComplete	This sample shows how to complete entries while the user types.
AutoSizeEdit	This sample shows how to resize the editor while the user types. This sample uses the C1FlexGrid control.
BarChart	This sample shows how to draw bar charts using the grid cells. This sample uses the C1FlexGrid control.
Blinker	This sample shows how to use CellStyles to make cells blink.
BoundDelete	This sample shows how to delete rows from a bound FlexGrid. This sample uses the C1FlexGrid control.
BoundFinishEdit	This sample shows how to commit changes to a DataRow after editing. This sample uses the C1FlexGrid control.
BoundImageMap	This sample shows how to use the ImageMap property with a data-bound grid. This sample uses the C1FlexGrid control.
C1FlexGridPrintable	This sample provides a printable (via C1Preview) C1FlexGrid control. This sample uses the C1FlexGrid control and calls the C1.C1Preview namespace.
CellBorders	This sample shows how to use OwnerDraw to provide custom cell borders. This sample uses the C1FlexGrid control.
CellDataType	This sample shows how to assign cell types (and editors) on a per-cell basis. This sample uses the C1FlexGrid control.
CellLabelDelay	This sample shows how to show cell labels after a specified interval.
CellNotes	This sample attaches notes to cells on a C1FlexGrid and implements Excel-style cell notes using two classes: <b>CellNote</b> and <b>CellNoteManager</b> .
Classic (C1FlexGridClassic)	This sample shows a grid that derives from C1FlexGrid and provides an object model compatible with the VSFlexGrid control.

Sample	Description
ColumnEditor	This sample shows how to expose the FlexGrid design-time column editor in your controls. This sample uses the C1FlexGrid control.
ColumnFilters	This sample shows how to use the filtering feature in <b>C1FlexGrid</b> 2010. The sample shows how you can control and customize the grid's built-in filters using the <a href="#">AllowFiltering</a> property and the <a href="#">AfterFilter</a> event.
ColumnOrder	This sample shows how to position columns dynamically. This sample uses the C1FlexGrid control.
ColumnWidthTip	This sample shows how to show a ToolTip while the user resizes columns.
CrossTabs	This sample shows how to summarize and cross-reference data using PivotTables.
CustomData	This sample shows how to implement a custom data source object. This sample uses the C1FlexGrid control.
CustomDataMap	This sample shows how to customize the options in drop-down lists when using DataMaps. This sample uses the C1FlexGrid control.
CustomEditor	This sample shows how to implement your own cell editor. This sample uses the C1FlexGrid control.
CustomEditors	This sample shows how to use built-in, .NET, and custom edit controls with the C1FlexGrid.
CustomFilters	This sample shows how to implement custom filters for the <b>C1FlexGrid</b> .
CustomMerge	This sample shows how to implement your own merging logic to create a TV-guide display. This sample calls the C1.Win.C1FlexGrid namespace.
CustomMerge2	This sample shows how to implement your own merging logic to mix merging modes. This sample calls the C1.Win.C1FlexGrid namespace.
CustomMerge3	This sample shows how to customize grouping by overriding the GetData method. This sample calls the C1.Win.C1FlexGrid namespace.
CustomMerge4	This sample shows how to customize grouping by overriding the GetData method. This sample calls the C1.Win.C1FlexGrid namespace.
CustomPrint	This sample shows how to use the CreateImage method to print grids with arbitrary row and column breaks. This sample uses the C1FlexGrid control.
CustomPrintMultiGrid	This sample shows how to use the CreateImage method to print multiple grids in a single document.
CustomSelection	This sample uses OwnerDraw to implement custom (non-rectangular) selection. This sample calls the C1.Win.C1FlexGrid namespace.
CustomSort	This sample shows how to implement your own sorting logic with a custom IComparer object. This sample uses the C1FlexGrid control.
DataIndex	This sample shows how to use the Row.DataIndex property to get the underlying data row. This sample uses the C1FlexGrid control.
DataTable	This sample shows how to bind to a DataTable, add and remove rows. This sample uses the C1FlexGrid control.
DataTree	This sample shows how to bind the grid to a hierarchical data source and show details in child grids. This sample calls the C1.Win.C1FlexGrid namespace.

Sample	Description
DBDynamicStyles	This sample shows how to assign styles to grid cells based on their contents. This sample uses the C1FlexGrid control.
DBImageField	This sample demonstrates how to show images stored in a DataTable. This sample uses the C1FlexGrid control.
DBImages	This sample shows how to bind the grid to a data base with image fields stored as OLE objects. This sample uses the C1FlexGrid control.
DBSchemaChange	This sample shows how to test grid response to changes in the data source object. This sample uses the C1FlexGrid control.
DBTree	This sample shows how to display bound data in a hierarchical tree view. This sample uses the C1FlexGrid control.
DBUpdate	This sample shows how to save changes to the underlying database. This sample uses the C1FlexGrid control.
DragDrop	This sample shows how to customize automatic Drag & Drop This sample uses the C1FlexGrid control.
DragImages	This sample shows how to customize Ole Drag and Drop to use cell images.
DragMerged	This sample shows how to drag groups of merged columns. This sample uses the C1FlexGrid control.
DragRow	This sample shows how to drag rows between grids. This sample uses the C1FlexGrid control.
DynamicStyles	This sample shows how to assign styles to a cell based on the contents of another cell. This sample uses the C1FlexGrid control.
ErrorInfo	This sample demonstrates how to show error information on the grid.
Excel	This sample shows how to load and save Microsoft Excel files (.xls).
ExcelStyleFilter	This sample adds a context-menu that allows sorting and filtering data.
FilterRow	This sample shows how to implement a FilterRow on a FlexGrid control. This sample uses the C1FlexGrid control.
FindRow	This sample shows how to how to find a row in the underlying datasource. This sample uses the C1FlexGrid control.
FlexByRow	This sample shows how to set up a grid in 'transposed' format. This sample uses the C1FlexGrid control.
FlexGroup	This sample shows how to implement Outlook-style grouping using the C1FlexGrid. This sample uses the C1FlexGrid control.
FlexHierarchical	This sample shows how to implement hierarchical data binding using the C1FlexGrid. This sample calls the C1.Win.C1FlexGrid namespace.
FreezeBottom	This sample demonstrates how to show frozen rows at the bottom of a grid. This sample uses the C1FlexGrid control.
GridChart	This sample shows how to attach a FlexGrid to a C1Chart control. This sample uses the C1FlexGrid and C1Chart controls.
HierData	This sample shows how to bind to hierarchical data sources (master-detail style). This

Sample	Description
	sample uses the C1FlexGrid control.
HierData2	This sample shows how to bind to hierarchical data sources (master-detail style).
HostControls	This sample shows how to host controls in grid cells. This sample uses the C1FlexGrid control.
Hyperlink	This sample shows how to add hyperlinks to cells in the FlexGrid. This sample uses the C1FlexGrid control.
LegacyDates	This sample shows how to use an unbound column to convert dates stored as strings into real dates.
LockedColumns	This sample shows how to prevent the user from selecting certain columns.
MasterDetail	This sample shows how to create and bind master-detail DataSets in code. This sample calls the C1.Win.C1FlexGrid namespace.
MergeStyles	This sample shows how to merge CellStyles assigned to rows, columns, and cells. This sample uses the C1FlexGrid control.
MultiColumnDictionary	This sample shows how to use the MultiColumnDictionary class to implement multi-column drop-downs.
MultiParent	This sample shows how to create a complex master-detail display.
MultiSelection	This sample shows how to implement multi-range selection.
NewFeatures20091	This sample shows the new features that were added for the V1/2009 release.
NewRowTemplate	This sample shows how to place a new row template above the grid.
Outline	This sample shows how to hide repeated data in an outline. This sample uses the C1FlexGrid control.
OwnerDraw	This sample shows how to autosize Owner-Drawn cells.
OwnerDrawAlpha	This sample shows how to use OwnerDraw with transparency to get MediaPlayer-like effects.
PasswordChar	This sample shows how to use the C1FlexGrid to enter and edit passwords. This sample uses the C1FlexGrid control.
PdfExport	This sample shows how to use the C1Pdf method to export C1FlexGrids to PDF files.
RowStateDisplay	This sample shows how to use different styles to display DataRow in different states. This sample uses the C1FlexGrid control.
RTFGrid	This sample shows how to display RTF text in grid cells. This sample uses the C1FlexGrid control.
ScrollPosition	This sample shows how to shows how the ScrollPosition property works. This sample uses the C1FlexGrid control.
SelectionMode	This sample shows how to shows the effect of the SelectionMode property. This sample uses the C1FlexGrid control.
SetupEditor	This sample shows how to use the SetupEditor event to customize the grid editors. This sample uses the C1FlexGrid control.
SimpleCalc	This sample uses the DataTable.Compute method to evaluate expressions in grid cells

Sample	Description
	and displays the results using OwnerDraw.
Sorting	This sample shows how to shows several sorting techniques. This sample uses the C1FlexGrid control.
SortNulls	This sample shows how to use a custom sort to place null values at the bottom of the grid.
Splits	This sample shows how to split a C1FlexGrid into multiple views. This sample uses the C1FlexGrid control.
SqlBuilder	This sample shows how to a grid-based graphical interface for creating SQL statements.
StartEditing	This sample shows how to implement a grid that stays in cell-editing mode. This sample uses the C1FlexGrid control.
Subtotals	This sample shows how to create subtotals for multiple columns. This sample uses the C1FlexGrid control.
Themes	This sample shows how to change the appearance of the grid to achieve one of several pre-defined looks.
ThreadedUpdate	This sample shows how to update the grid from a different thread.
TreeCheck	This sample shows how to add checkboxes to a grid tree. This sample uses the C1FlexGrid control.
TreeNode	This sample shows how to manage an outline tree using the FlexGrid Node objects. This sample uses the C1FlexGrid control.
TriStateBound	This sample shows how to provide three-state (grayable) values in Boolean columns. This sample calls the C1.Win.C1FlexGrid namespace.
UnboundColumns	This sample shows how to additional support for unbound columns.
ZoomCell	This sample shows how to zoom in on the selected cell.

## C1FlexGrid for Mobile Devices Visual Basic Samples

Sample	Description
Analyze	This sample shows how to provide dynamic data sorting and grouping. This sample uses the C1FlexGrid control.
BarChart	This sample shows how to draw bar charts using the grid cells. This sample uses the C1FlexGrid control.
EditData	This sample demonstrates how to show different types of built-in editors and dynamic formatting. This sample uses the C1FlexGrid control.
ThreadedUpdate	This sample shows how to update the grid from a different thread.

## C1FlexGrid for Mobile Devices C# Samples

Sample	Description
Analyze	This sample shows how to provide dynamic data sorting and grouping. This sample uses the C1FlexGrid control.



Sample	Description
BarChart	This sample shows how to draw bar charts using the grid cells. This sample uses the C1FlexGrid control.
ContextMenu	This sample demonstrates how to show a ContextMenu after selecting an item on the grid.
DataTree	This sample demonstrates how to show hierarchical data in a drill-down grid.
EditData	This sample demonstrates how to show different types of built-in editors and dynamic formatting.
MasterDetail	This sample demonstrates how to show hierarchial data using two grids.
TransparentImages	This sample shows how to render images with transparency.

## FlexGrid for WinForms Tutorials

The following sections contain tutorials that illustrate some of the main features in the [C1FlexGrid](#) control. The tutorials walk you through the creation of several simple projects, describing each step in detail. The distribution CD contains more sophisticated samples that can be used as a reference.

The tutorials are:

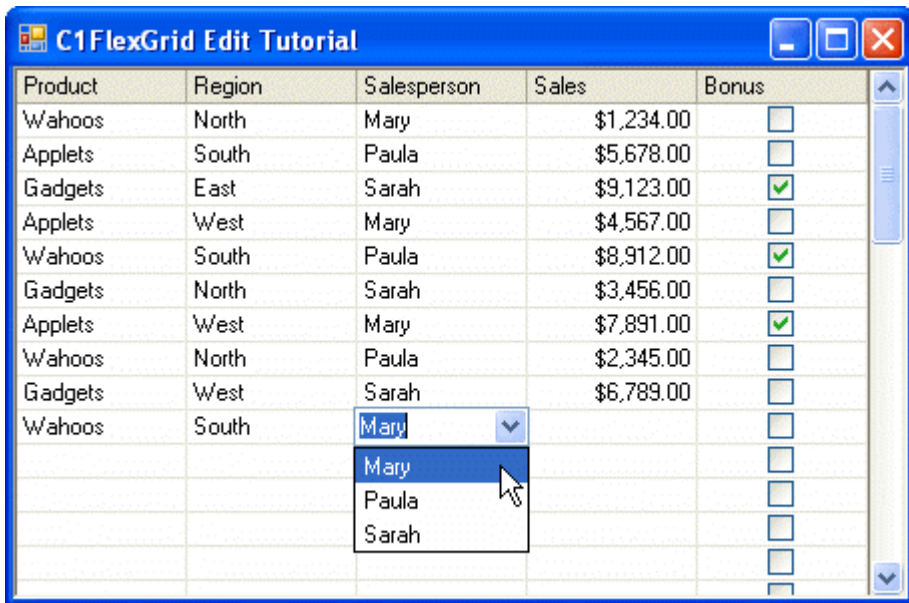
Tutorial	Description
<a href="#">Edit Tutorial</a>	Starting with a basic data-entry grid, this tutorial shows how to implement data formatting, check boxes, drop-down lists, input masks, data validation, and clipboard support.
<a href="#">Outline Tutorial</a>	Shows how you can use the <a href="#">C1FlexGrid</a> as an outliner to display structured (or hierarchical) data.
<a href="#">Data Analysis Tutorial</a>	Starting with a grid containing sales data for different products, regions, and salespeople, this tutorial show how to implement dynamic layout (column order), automatic sorting, cell merging, automatic subtotals, and outlining.

## Edit Tutorial

This tutorial starts with a basic data-entry grid, then adds the following features:

1. Formatting
2. Check boxes
3. Drop-down lists
4. Complex data validation
5. Clipboard support
6. Custom editors

Here is what the final application will look like:



 **Note:** A video is available for this tutorial on the [GrapeCity Videos](#) Web page.

## Step 1 of 6: Create the C1FlexGrid Control for the Edit Tutorial

Start a new project and add a [C1FlexGrid](#) control to the form by clicking the **C1FlexGrid** icon on the Toolbox, then clicking on the form and dragging until the object has the proper size.

If you can't find the C1FlexGrid control in the toolbox, right-click on the toolbox and select **Choose Items**. Then, look for the C1FlexGrid control on the list of .NET components and make sure it is checked. If you can't find the grid in the component list, you may need to re-install the product.

1. Set the following properties in the Properties window for the C1FlexGrid control:

Property	Setting
Dock	Fill
Cols.Count	5
Cols.Fixed	0

2. Double-click the form caption area to open the code window. At the top of the file, add the following statement:

### To write code in Visual Basic

```
Visual Basic
Imports C1.Win.C1FlexGrid
```

### To write code in C#

```
C#
using C1.Win.C1FlexGrid;
```

This makes the objects defined in the C1FlexGrid assembly visible to the project and saves a lot of typing.

3. Set up the columns by either in the designer through the **Column Tasks** menu or the **C1FlexGrid Column Editor**, or in code.

### In the Designer

- o Select the first column in the grid. This will open the **Column Tasks** menu for the column.
- o Enter Product in the Column Caption and Data Field boxes.
- o Set the **Column Caption** and **Data Field** boxes for the remainder of the columns as follows:

Column 1	
Column Caption	Region
Data Field	Region
Column 2	
Column Caption	Salesperson
Data Field	Salesperson
Column 3	
Column Caption	Sales
Data Field	Sales
Column 4	
Column Caption	Bonus
Data Field	Bonus

Alternatively, the columns can also be set up through the **C1FlexGrid Column Editor**:

- o Open the **C1FlexGrid Column Editor** by clicking **Designer** in the **C1FlexGrid Tasks** menu. For details on how to access the **C1FlexGrid Column Editor**, see [Accessing the C1FlexGrid Column Editor](#).
- o Select the Column 0 in the right pane.
- o In the left pane, set the **Name** and **Caption** properties to **Product**.
- o Set the **Name** and **Caption** properties for the remainder of the columns as follows:

Column 1	
Name	Region
Caption	Region
Column 2	
Name	Salesperson
Caption	Salesperson
Column 3	
Name	Sales
Caption	Sales

Column 1	
Column 4	
Name	Bonus
Caption	Bonus

- o Click **OK** when finished to close the editor.

### In Code

Add the following code to the **Form\_Load** event:

#### To write code in Visual Basic

```

Visual Basic

' Set up the columns.
Dim cols As String = "Product|Region|Salesperson|Sales|Bonus"
Dim colNames As String() = cols.Split("|")
Dim i%For i = 0 To C1FlexGrid1.Cols.Count - 1
    C1FlexGrid1(0, i) = colNames(i)
    C1FlexGrid1.Cols(i).Name = colNames(i)Next
    
```

#### To write code in C#

```

C#

// Set up the columns.
string cols = "Product|Region|Salesperson|Sales|Bonus";
string[] colNames = cols.Split(new char[] { '|' });
for (int i = 0; i <= this.c1FlexGrid1.Cols.Count - 1; i++)
{
    c1FlexGrid1[0, i] = colNames[i];
    c1FlexGrid1.Cols[i].Name = colNames[i];
}
    
```

## Run the program and observe the following:

That's it. Press F5 to run the project, and you can start typing data into the control. Press F2 or the spacebar to edit existing entries, or just type new entries over existing ones.



## To write code in Visual Basic

### Visual Basic

```
' Set the column DataType and Format.
Dim c As Column = C1FlexGrid1.Cols("Sales")
c.DataType = GetType(Decimal)

' Currency.
c.Format = "c2"
c = C1FlexGrid1.Cols("Bonus")
c.DataType = GetType(Boolean)
c.ImageAlign = ImageAlignEnum.CenterCenter
```

## To write code in C#

### C#

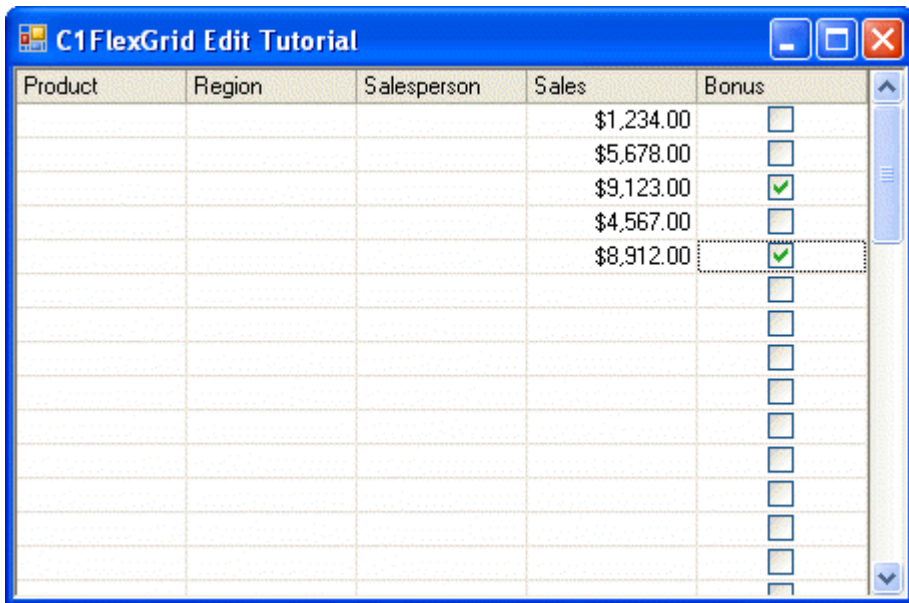
```
// Set the column DataType and Format.
Column c = c1FlexGrid1.Cols["Sales"];
c.DataType = typeof(Decimal);

// Currency.
c.Format = "c2";
c = c1FlexGrid1.Cols["Bonus"];
c.DataType = typeof(bool);
c.ImageAlign = ImageAlignEnum.CenterCenter;
```

## Run the program and observe the following:

The new code formats the *Sales* column to store and display currency values, and the *Bonus* column to deal with Boolean values.

If you enter some numeric and non-numeric values in the *Sales* column, you will notice that the grid won't accept non-numeric entries. The *Bonus* column displays values as checkboxes, which can be toggled with the mouse or with the keyboard. This is the default behavior for Boolean columns.



Note that the Format property does not affect the data in any way, only how it is displayed.

## Step 3 of 6: Incorporate Drop-Down Lists

Entering data is a tedious and error-prone process. Drop-down lists are great because they minimize the amount of typing you must do, reduce the chance of errors, and increase the consistency of the data.

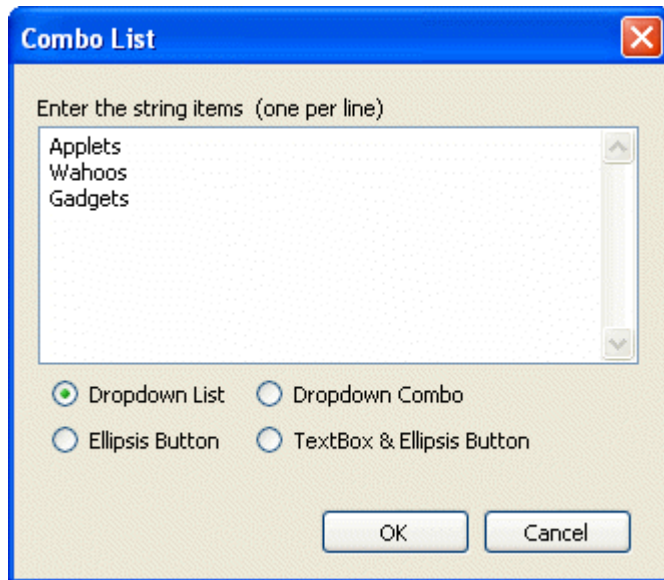
Let's assume that our tutorial project only involves sales of three products (Applets, Wahoos, and Gadgets), in four regions (North, South, East, and West), and that there are three full-time sales people (Mary, Sarah, and Paula).

To use these lists as drop-downs in the [C1FlexGrid](#) control, all you have to do is build strings containing the items in each list (separated by pipe characters) and assign them to the [ComboList](#) property of each column. This property can be set either in the designer or in code.

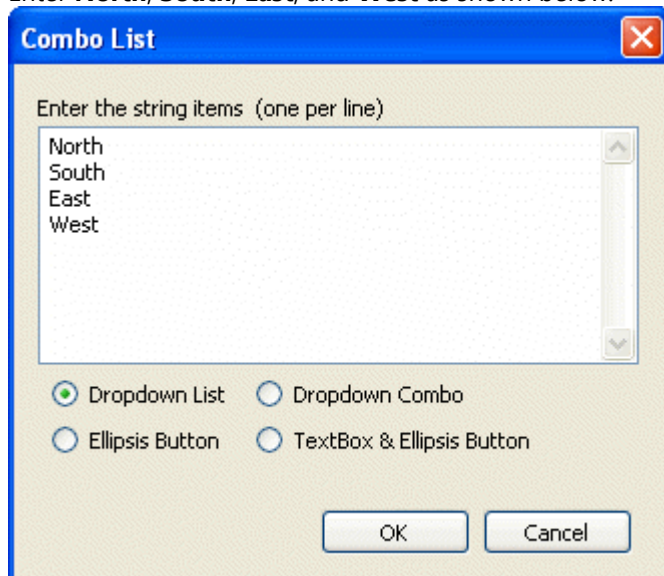
### In the Designer

1. Select the *Product* column. This will open the **Column Tasks** menu for the *Product* column.
2. Click the **ellipsis** button in the **Combo List** box to open the **Combo List** dialog box.
3. Enter **Applets, Wahoos, and Gadgets** as shown below:

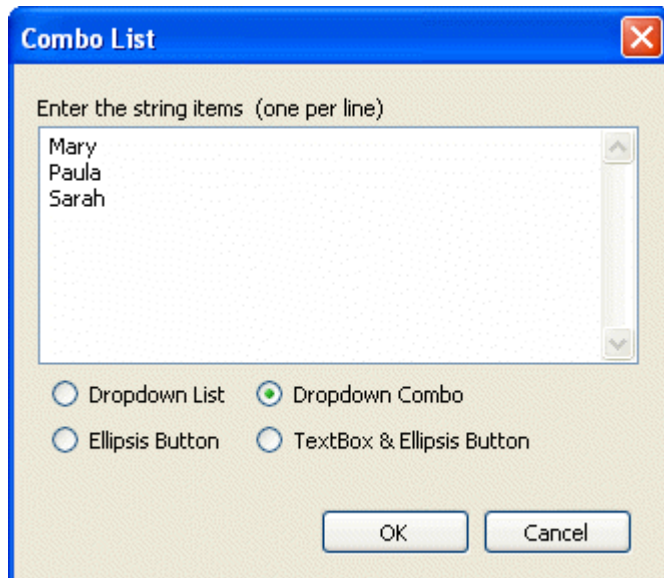




4. Click **OK** to close the **Combo List** dialog box.
5. Select the *Region* column.
6. Click the **ellipsis** button in the **Combo List** box.
7. Enter **North**, **South**, **East**, and **West** as shown below:



8. Click **OK**.
9. Select the *Salesperson* column.
10. Click the **ellipsis** button in the **Combo List** box.
11. Enter **Mary**, **Paula**, and **Sarah** as shown below:



12. Select the **Dropdown Combo** option.
13. Click **OK**.

Alternatively, the `ComboList` property can be set using the **C1FlexGrid Column Editor**:

1. Open the **C1FlexGrid Column Editor** by selecting **Designer** in the **C1FlexGrid Tasks** menu. For details on how to access the **C1FlexGrid Column Editor**, see [Accessing the C1FlexGrid Column Editor](#).
2. Select the `Product` column in the right pane.
3. In the left pane, set the **ComboList** property to **Applets|Wahoos|Gadgets**.
4. Select the `Region` column in the right pane.
5. In the left pane, set the **ComboList** property to **North|South|East|West**.
6. Select the `Salesperson` column in the right pane.
7. In the left pane set the **ComboList** property to **|Mary|Paula|Sarah**.
8. Click **OK** to close the editor.

## In Code

To add the combolists, add the following code after the code added in [Step 2 of 6: Set Column Types and Formats](#):

### To write code in Visual Basic

Visual Basic

```
' Set up the drop-down lists.C1FlexGrid1.
Cols("Product").ComboList = "Applets|Wahoos|Gadgets"
C1FlexGrid1.Cols("Region").ComboList = "North|South|East|West"
C1FlexGrid1.Cols("Salesperson").ComboList = "|Mary|Paula|Sarah"
```

### To write code in C#

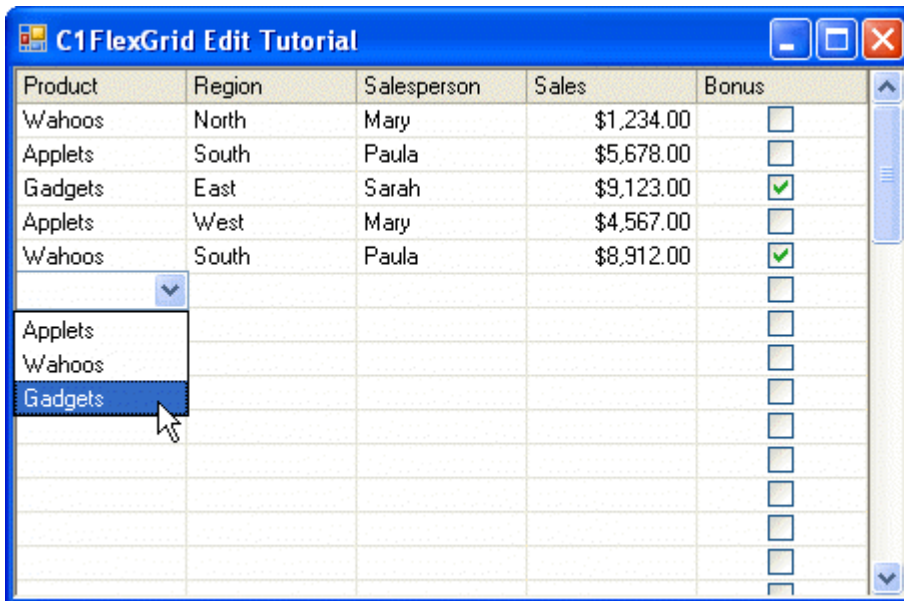
C#

```
// Set up the drop-down lists.
c1FlexGrid1.Cols["Product"].ComboList = "Applets|Wahoos|Gadgets";
c1FlexGrid1.Cols["Region"].ComboList = "North|South|East|West";
c1FlexGrid1.Cols["Salesperson"].ComboList = "|Mary|Paula|Sarah";
```

## Run the program and observe the following:

Notice how the last ComboBox string starts with a pipe (text input cursor). This allows users to type additional names that are not on the list. In other words, these values will be edited using a drop-down combo, as opposed to a simple drop-down list.

Press F5 to run the project again, and then move the cursor around. When you move the cursor to one of the columns that have combo lists, a drop-down button becomes visible. You may click on it to show the list, or simply type the first letter of an entry to highlight it on the list.



## Step 4 of 6: Add Data Validation

If you assign a data type to a grid column, the grid will ensure that only data of the proper type is stored in that column. This helps prevent errors, but you will often need stricter validation to ensure that the data entered is correct. For that, you should use the [ValidateEdit](#) event.

For example, imagine that anti-trust regulations prevent us from selling our Applets in the North region. To prevent data-entry mistakes and costly lawsuits, we want to prevent users from entering this combination into the control.

The following event checks the data after each edit and prevents invalid entries. Add the following code to the form:

### To write code in Visual Basic

#### Visual Basic

```
Private Sub C1FlexGrid1_ValidateEdit(ByVal sender As Object, ByVal e As
ValidateEditEventArgs) Handles C1FlexGrid1.ValidateEdit
    Dim rgn As String = String.Empty
    Dim prd As String = String.Empty

    ' Collect the data we need.
    Select Case e.Col
        Case 0
            prd = C1FlexGrid1.Editor.Text
            rgn = C1FlexGrid1(e.Row, "Region")
```

```

        Case 1
            prd = C1FlexGrid1(e.Row, "Product")
            rgn = C1FlexGrid1.Editor.Text
        End Select

        ' We can't sell Applets in the North Region.>
    If prd = "Applets" And rgn = "North" Then
        MsgBox("Warning: Regulation #12333AS/SDA-23 forbids " & _
            "the sale of " & prd & " in region " & rgn & ". " & _
            "Please verify input.")
        e.Cancel = True
    End If
End Sub

```

### To write code in C#

```

C#
private void c1FlexGrid1_ValidateEdit( object sender, ValidateEditEventArgs e)
{
    string rgn = string.Empty;
    string prd = string.Empty;

    // Collect the data we need.
    switch (e.Col)
    {
        case 0:
            prd = c1FlexGrid1.Editor.Text;
            rgn = (string)c1FlexGrid1[e.Row, "Region"];
            break;
        case 1:
            prd = (string)c1FlexGrid1[e.Row, "Product"];
            rgn = c1FlexGrid1.Editor.Text;
            break;
    }

    // We can't sell Applets in the North Region.
    if ( prd == "Applets" && rgn == "North" )
    {
        MessageBox.Show("Warning: Regulation #12333AS/SDA-23 forbids " +
            "the sale of " + prd + " in region " + rgn + ". " +
            "Please verify input.");
        e.Cancel = true;
    }
}

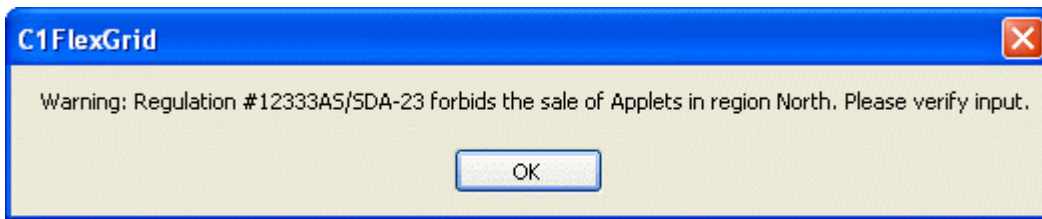
```

### Run the program and observe the following:

The function starts by gathering the input that needs to be validated. Note that the values being checked are retrieved using the **Editor.Text** property. This is necessary because the edits have not yet been applied to the control. If the test fails, the function displays a warning and then sets the *Cancel* parameter to **True**, which cancels the edits and puts the

cell back in edit mode so the user can try again.

Press F5 to run the project again, then try inputting some bad values. You will see that the control will reject them.



## Step 5 of 6: Add Clipboard Support

The Windows clipboard is a very useful device for transferring information between applications.

Adding clipboard support to **FlexGrid for WinForms** projects is fairly easy. Simply set the [AutoClipboard](#) property to **True** either in the designer or in code, and the grid will automatically handle all standard keyboard commands related to the clipboard: CTRL+X or SHIFT+DELETE to cut, CTRL+C or CTRL+INSERT to copy, and CTRL+V or SHIFT+INSERT to paste.

### In the Designer

Locate the **AutoClipboard** property in the Properties window and set it to **True**.

### In Code

Add the following code after the code added in [Step 3 of 6: Incorporate Drop-Down Lists](#):

#### To write code in Visual Basic

```
Visual Basic  
C1FlexGrid1.AutoClipboard = True
```

#### To write code in C#

```
C#  
c1FlexGrid1.AutoClipboard = true;
```

Another great Windows feature that is closely related to clipboard operations is OLE Drag and Drop. [C1FlexGrid](#) has two properties, [DragMode](#) and [DropMode](#), which help implement this feature. Just set both properties to their automatic settings either in the designer or in code, and you will be able to drag selections by their edges and drop them into other applications such as Microsoft Excel, or drag ranges from an Excel spreadsheet and drop them into the [C1FlexGrid](#) control.

### In the Designer

Locate the **DragMode** and **DropMode** properties and set them both to **Automatic**.

### In Code

Add the following code after the code to set the **AutoClipboard** property:

#### To write code in Visual Basic

##### Visual Basic

```
C1FlexGrid1.DragMode = DragModeEnum.Automatic
C1FlexGrid1.DropMode = DropModeEnum.Automatic
```

#### To write code in C#

##### C#

```
c1FlexGrid1.DragMode = DragModeEnum.Automatic;
c1FlexGrid1.DropMode = DropModeEnum.Automatic;
```

## Run the program and observe the following:

Press F5 to run the project again, then try copying and pasting some data. Note that you can paste invalid data, because our paste code does not perform any data validation. This is left as an exercise for the reader.

## Step 6 of 6: Include Custom Editors

The **C1FlexGrid** has powerful built-in editors for entering text, masked text, selecting from lists, checkboxes, and more. But in some cases you may want to use a custom editor instead, perhaps one of the input controls in the **C1Input** library or a control that you wrote. Starting in version 2.5, the FlexGrid allows you to easily plug-in custom editors.

To attach a custom editor to the *Sales* column on our tutorial, start by adding a **NumericUpDown** control to the form.

1. Set the following properties the Properties window to for the **NumericUpDown** control:

Property	Setting
BorderStyle	None
DecimalPlaces	2
Maximum	5000000
ThousandsSeparator	True
Visible	False

2. Select **Designer** from the **C1FlexGrid Tasks** menu to bring up the grid's column editor.
3. Select the *Sales* column (or Column 3 where the Sales figures will be), and set the **Editor** property to **NumericUpDown1**.

## Run the program and observe the following:

Now run the project and try editing some values in the *Sales* column. Notice that the grid uses the **NumericUpDown** control instead of the built-in editors. The control is properly positioned, initialized, and managed. When you move the focus to a different cell, the value is stored in the grid.

Product	Region	Salesperson	Sales	Bonus
Wahoos	North	Mary	\$1,234.00	<input type="checkbox"/>
Applets	South	Paula	\$5,678.00	<input type="checkbox"/>
Gadgets	East	Sarah	\$9,123.00	<input checked="" type="checkbox"/>
Applets	West	Mary	\$4,567.00	<input type="checkbox"/>
Wahoos	South	Paula	\$8,912.00	<input checked="" type="checkbox"/>
Gadgets	North	Sarah	8,914.00	<input type="checkbox"/>
				<input type="checkbox"/>
				<input type="checkbox"/>
				<input type="checkbox"/>
				<input type="checkbox"/>
				<input type="checkbox"/>
				<input type="checkbox"/>
				<input type="checkbox"/>
				<input type="checkbox"/>
				<input type="checkbox"/>

But the custom editor doesn't behave exactly like the built-in ones. For example:

1. When you start editing by typing a number, the old value isn't cleared.
2. The size of the editor isn't exactly right (it looks a bit too small).
3. There's a beep when you press ENTER to finish editing.

You can overcome these problems in a couple of ways. One way would be to use the editor's events, but that would make it difficult to reuse the control in other projects. Another way would be to create a derived class and implement some methods in the **IC1EmbeddedEditor** interface, as in the following code.

#### To write code in Visual Basic

##### Visual Basic

```
Public Class MyUpDown
    Inherits NumericUpDown

    ' Set the initial value.
    Public Sub C1EditorInitialize(ByVal value As Object, ByVal editorAttributes As
IDictionary)

        ' Apply the initial value.
        value = Convert.ChangeType(value, GetType(Decimal))

        ' Select the whole entry.
        MyBase.Select(0, Int32.MaxValue)
    End Sub

    ' Set the FontHeight so the control honors the rectangle height.
    Public Sub C1EditorUpdateBounds(ByVal rc As Rectangle)
        MyBase.FontHeight = rc.Height
        Bounds = rc
    End Sub

    ' Suppress the beeps when a user presses ENTER.
```

```

Protected Overrides Function ProcessDialogKey(ByVal keyData As Keys) As Boolean
    If (keyData = Keys.Enter) Then
        Parent.Focus()
        If (Parent.Focused) Then SendKeys.Send("{Down}")
        Return True
    End If
    Return MyBase.ProcessDialogKey(keyData)
End Function
End Class

```

### To write code in C#

```

C#
internal class MyUpDown : NumericUpDown
{
    // Set the initial value.
    public void C1EditorInitialize(object value, System.Collections.IDictionary
editorAttributes)
    {
        // Apply the initial value.
        Value = (decimal)Convert.ChangeType(value, typeof(decimal));

        // Select the whole entry.
        Select(0, int.MaxValue);
    }

    // Set the FontHeight so the control honors the rectangle height.
    public void C1EditorUpdateBounds(Rectangle rc)
    {
        base.FontHeight = rc.Height;
        Bounds = rc;
    }

    // Suppress the beeps when a user presses ENTER.
    override protected bool ProcessDialogKey(Keys keyData)
    {
        if (keyData == Keys.Enter)
        {
            Parent.Focus();
            if (Parent.Focused) SendKeys.Send("{Down}");
            return true;
        }
        return base.ProcessDialogKey(keyData);
    }
}

```

The previous code implements three methods:

- **C1EditorInitialize** is called to initialize the editor. It sets the initial value and then selects the whole entry. This will take care of the first problem. Because the whole entry is selected, typing the first character will now replace the current contents as we want.
- **C1EditorUpdateBounds** is called to position the editor over the cell being edited. The height of the



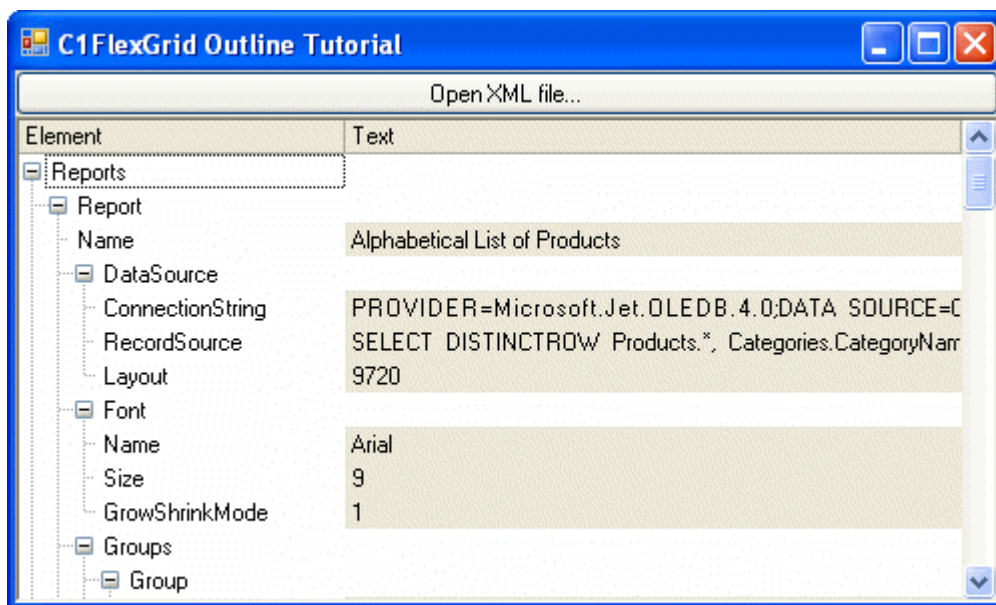
**NumericUpDown** control is set automatically based on the font size, though (that is why it looks too short for the cell). The code sets the editor's **FontHeight** property so it will be positioned exactly over the cell.

- **ProcessDialogKey** method is overridden to suppress the beeps when the user presses the Enter key.

## Outline Tutorial

This tutorial shows how you can use the [C1FlexGrid](#) as an outliner to display structured (or hierarchical) data. When used as an outliner, the C1FlexGrid control behaves like a Tree control, displaying nodes that can be collapsed or expanded to show branches containing subordinate data.

The tutorial allows you to load an XML document into the grid, where it is displayed as a tree. Here is what the final application will look like:



## Step 1 of 5: Create the Controls

Start a new project and add two controls:

- A command Button near the top of the form.
- A [C1FlexGrid](#) control in the area below the button.

If you can't find the C1FlexGrid control in the Toolbox, right-click on the toolbox and select **Choose Items**. Then, look for the C1FlexGrid control on the list of .NET components and make sure it is checked. If you can't find the grid in the component list, you may need to re-install the product.

1. Set the following properties in the Properties window:

- Command Button

Property	Setting
Dock	Top
Text	"Open XML File..."

- [C1FlexGrid](#)

Property	Setting
Dock	Fill

- Double-click the form caption area to open the code window. At the top of the file, add the following statement:

### To write code in Visual Basic

```
Visual Basic
Imports Cl.Win.C1FlexGrid
```

### To write code in C#

```
C#
using Cl.Win.C1FlexGrid;
```

This makes the objects defined in the C1FlexGrid assembly visible to the project and saves a lot of typing.

- Set up the grid either in the designer using the Properties window and editors, or in code by typing (or copying) the following code.

### In the Designer

Set the following properties for the C1FlexGrid control in the Properties window:

Property	Setting
Cols.Count	2
Cols.Fixed	0
ExtendLastCol	True
Rows.Count	1
Tree.Column	0
Tree.Style	SimpleLeaf

Set up the styles for the grid:

- Open the **C1FlexGrid Style Editor** by selecting **Styles** from the **C1FlexGrid Tasks** menu. For more information on accessing the **C1FlexGrid Style Editor**, see [Accessing the C1FlexGrid Style Editor](#).
- Select Normal in the list of **Built-In Styles**.
- Set the **Border.Style** property to **None**, the **TextAlign** property to **LeftCenter**, and the **WordWrap** property to **False**.
- Click the **Add** button.
- Rename CustomStyle1 to Data.
- Set the **BackColor** property to **Control**.
- Click **OK** to close the editor.

Set up the columns for the grid:

- Select Column 0 in the grid. This will open the **Column Tasks** menu for Column 0.
- Set the **Column Caption** to **Element**.
- Uncheck the **Allow Editing** check box.
- Select Column 1.

- Set the **Column Caption** to **Text**.

Alternatively, the columns can also be set up through the **C1FlexGrid Column Editor**:

- Open the **C1FlexGrid Column Editor** by selecting **Designer** in the **C1FlexGrid Tasks** menu. For details on how to access the **C1FlexGrid Column Editor**, see [Accessing the C1FlexGrid Column Editor](#).
- Select Column 0 in the right pane.
- In the left pane, set the **AllowEditing** property to **False** and the **Caption** property to **Element**.
- Select Column 1 in the right pane.
- In the left pane, set the **Caption** property to **Text**.
- Click **OK** to close the editor.

## In Code

### To write code in Visual Basic

#### Visual Basic

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load

    ' Initialize the grid.
    C1FlexGrid1.Rows.Count = 1
    C1FlexGrid1.Cols.Count = 2
    C1FlexGrid1.Cols.Fixed = 0
    C1FlexGrid1.ExtendLastCol = True
    C1FlexGrid1(0, 0) = "Element"
    C1FlexGrid1(0, 1) = "Text"

    ' Initialize the outline tree.
    C1FlexGrid1.Tree.Column = 0
    C1FlexGrid1.Tree.Style = TreeStyleFlags.SimpleLeaf
    C1FlexGrid1.Cols(0).AllowEditing = False

    ' Initialize styles.
    C1FlexGrid1.Styles.Normal.Border.Style = BorderStyleEnum.None
    C1FlexGrid1.Styles.Normal.TextAlign = TextAlignEnum.LeftCenter
    C1FlexGrid1.Styles.Normal.WordWrap = False
    Dim cs As CellStyle = C1FlexGrid1.Styles.Add("Data")
    cs.BackColor = SystemColors.Control

End Sub
```

### To write code in C#

#### C#

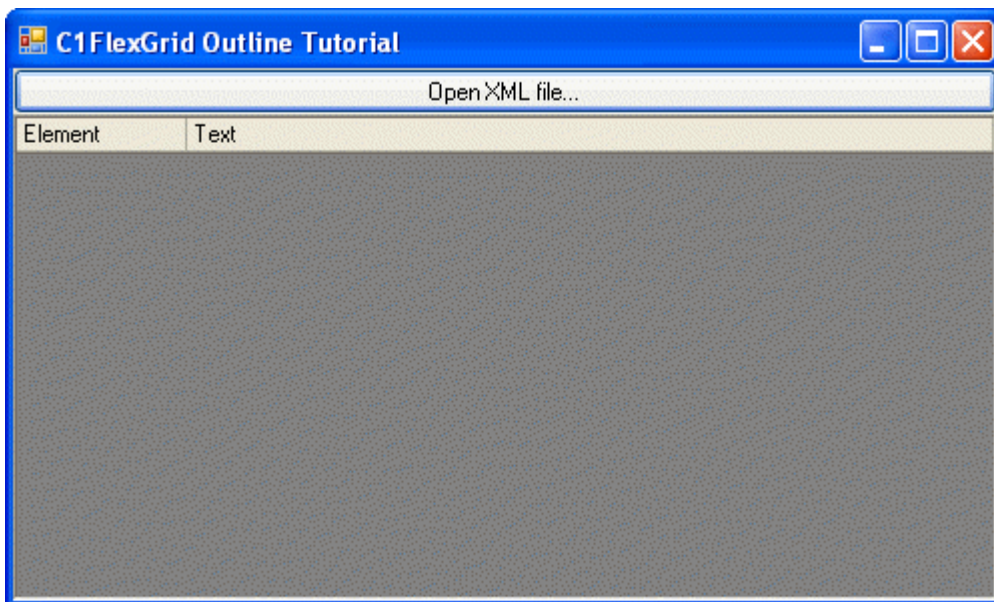
```
private void Form1_Load( System.object sender, System.EventArgs e)
{
    // Initialize the grid.
    c1FlexGrid1.Rows.Count = 1;
    c1FlexGrid1.Cols.Count = 2;
    c1FlexGrid1.Cols.Fixed = 0;
    c1FlexGrid1.ExtendLastCol = true;
    c1FlexGrid1[0, 0] = "Element";
    c1FlexGrid1[0, 1] = "Text";
}
```

```
// Initialize the outline tree.
c1FlexGrid1.Tree.Column = 0;
c1FlexGrid1.Tree.Style = TreeStyleFlags.SimpleLeaf;
c1FlexGrid1.Cols[0].AllowEditing = false;

// Initialize styles.
c1FlexGrid1.Styles.Normal.Border.Style = BorderStyleEnum.None;
c1FlexGrid1.Styles.Normal.TextAlign = TextAlignEnum.LeftCenter;
c1FlexGrid1.Styles.Normal.WordWrap = false;
CellStyle cs = c1FlexGrid1.Styles.Add("Data");
cs.BackColor = SystemColors.Control;}
```

## Run the program and observe the following:

The code starts by setting up the grid layout and column heading text.



Next, it initializes the outline tree using the [Tree](#) property and prevents editing of the XML nodes by setting the [AllowEditing](#) property of the first column to **False**. Note that the user can still edit data in the second column, which contains the data in each XML node.

Now the control is ready. We can start adding some code to it.

## Step 2 of 5: Read the Data and Build the Outline

To read the data and build the outline, add code to the **Button1\_Click** event and add the **GetXMLData** routine.

1. Double-click the command button and add the following code to the **Button1\_Click** event:

### To write code in Visual Basic

```
Visual Basic
```

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click

    ' Get the file name.
    Dim fo As OpenFileDialog = New OpenFileDialog()
    fo.DefaultExt = ".xml"
    fo.Filter = "XML Files (*.xml)|*.xml"
    If fo.ShowDialog() <> Windows.Forms.DialogResult.OK Then Exit Sub

    ' Load the XML file.
    Dim xdoc As System.Xml.XmlDocument = New System.Xml.XmlDocument()
    xdoc.Load(fo.FileName)

    ' Stop redrawing to improve speed.
    C1FlexGrid1.Redraw = False

    ' Populate the grid.
    C1FlexGrid1.Rows.Count = 1
    GetXMLData(xdoc.ChildNodes(1), 0)

    ' Autosize the tree column.
    C1FlexGrid1.AutoSizeCol(0)

    ' Show levels 0, 1, and 2.
    C1FlexGrid1.Tree.Show(2)

    ' Start redrawing.
    C1FlexGrid1.Redraw = True
End Sub
```

### To write code in C#

C#

```
private void button1_Click(System.Object sender, System.EventArgs e)
{
    // Get the file name.
    OpenFileDialog fo = new OpenFileDialog();
    fo.DefaultExt = ".xml";
    fo.Filter = "XML Files (*.xml)|*.xml";
    if ( fo.ShowDialog() != DialogResult.OK ) return;

    // Load the XML file.
    System.Xml.XmlDocument xdoc = new System.Xml.XmlDocument();
    xdoc.Load(fo.FileName);

    // Stop redrawing to improve speed.
    c1FlexGrid1.Redraw = false;

    // Populate the grid.
    c1FlexGrid1.Rows.Count = 1;
    GetXMLData(xdoc.ChildNodes[1], 0);
}
```

```
// Autosize the tree column.
c1FlexGrid1.AutoSizeCol(0);

// Show levels 0, 1, and 2.
c1FlexGrid1.Tree.Show(2);

// Start redrawing.
c1FlexGrid1.Redraw = true;
}
```

## Observe the following:

The routine starts by showing an **OpenFileDialog** that allows the user to select an XML file to load into the grid. When the file is selected, the routine loads it into an **XmlDocument** object, which parses the contents of the file into memory.

The routine then sets the grid's **Redraw** property to **False** to suspend repainting while the control is populated. This technique improves performance significantly, and you should always use it when adding substantial amounts of data to the **C1FlexGrid**.

Next, the routine clears any data by setting **Count** to 1, and calls the **GetXMLData** routine to populate the control with the contents of the **XmlDocument**. The **GetXMLData** routine is the main one in this tutorial, and is listed below.

After the grid has been populated, the routine uses the **AutoSizeCol** method to adjust the width of the first column based on its contents, and the **Show** method to expand the outline and show levels 0, 1, and 2. The routine then sets the **Redraw** property back to **True** so the grid starts repainting normally.

2. The **GetXMLData** routine is the most interesting one in this tutorial. It traverses the **XMLDocument** object and builds the outline tree. Add the following code to the form:

### To write code in Visual Basic

#### Visual Basic

```
Private Sub GetXMLData(ByVal node As System.Xml.XmlNode, ByVal level As Integer)
    ' Skip the comment nodes.
    If node.NodeType = System.Xml.XmlNodeType.Comment Then
        Exit Sub
    End If

    ' Add a new row for this node.
    Dim row As Integer = C1FlexGrid1.Rows.Count
    C1FlexGrid1.Rows.Add()

    ' Add data to the new row.
    C1FlexGrid1(row, 0) = node.Name
    If node.ChildNodes.Count = 1 Then
        C1FlexGrid1(row, 1) = node.InnerText
        C1FlexGrid1.SetCellStyle(row, 1, C1FlexGrid1.Styles("Data"))
    End If

    ' If the node has a "Name" subnode, save it to use as a ToolTip.
    If node.ChildNodes.Count > 0 Then
```

```

    Dim ndName As System.Xml.XmlNode = node.SelectSingleNode("Name")
    If Not (ndName Is Nothing) Then
        c1FlexGrid1.Rows(row).UserData = ndName.InnerText
    End If
End If

' If this node has children, get them as well.
If node.ChildNodes.Count > 1 Then

    ' Make this row a node.
    c1FlexGrid1.Rows(row).IsNode = True
    c1FlexGrid1.Rows(row).Node.Level = level

    ' Recurse to get children.
    Dim child As System.Xml.XmlNode
    For Each child In node.ChildNodes
        GetXMLData(child, level + 1)
    Next
End If
End Sub

```

**To write code in C#**

```

C#
private void GetXMLData(System.Xml.XmlNode node, int level)
{
    // Skip the comment nodes.
    if ( node.NodeType == System.Xml.XmlNodeType.Comment )
    {
        return;
    }

    // Add a new row for this node.
    int row = c1FlexGrid1.Rows.Count;
    c1FlexGrid1.Rows.Add();

    // Add data to the new row.
    c1FlexGrid1[row, 0] = node.Name;
    if ( node.ChildNodes.Count == 1 )
    {
        c1FlexGrid1[row, 1] = node.InnerText;
        c1FlexGrid1.SetCellStyle(row, 1, c1FlexGrid1.Styles["Data"]);
    }

    // If the node has a "Name" subnode, save it to use as a Tooltip.
    if (node.ChildNodes.Count > 0)
    {
        System.Xml.XmlNode ndName = node.SelectSingleNode("Name");
        if (ndName != null)
        {
            c1FlexGrid1.Rows[row].UserData = ndName.InnerText;
        }
    }
}

```

```
    }  
  }  
  
  // If this node has children, get them as well.  
  if ( node.ChildNodes.Count > 1 )  
  {  
    // Make this row a node.  
    c1FlexGrid1.Rows[row].IsNode = true;  
    c1FlexGrid1.Rows[row].Node.Level = level;  
  
    // Recurse to get children.  
    foreach (System.Xml.XmlNode child in node.ChildNodes)  
      GetXMLData(child, level + 1);  
  }  
}
```

## Observe the following:

The routine starts by skipping XML comment nodes. Then it uses the **Rows.Add** method to add a new row to the grid.

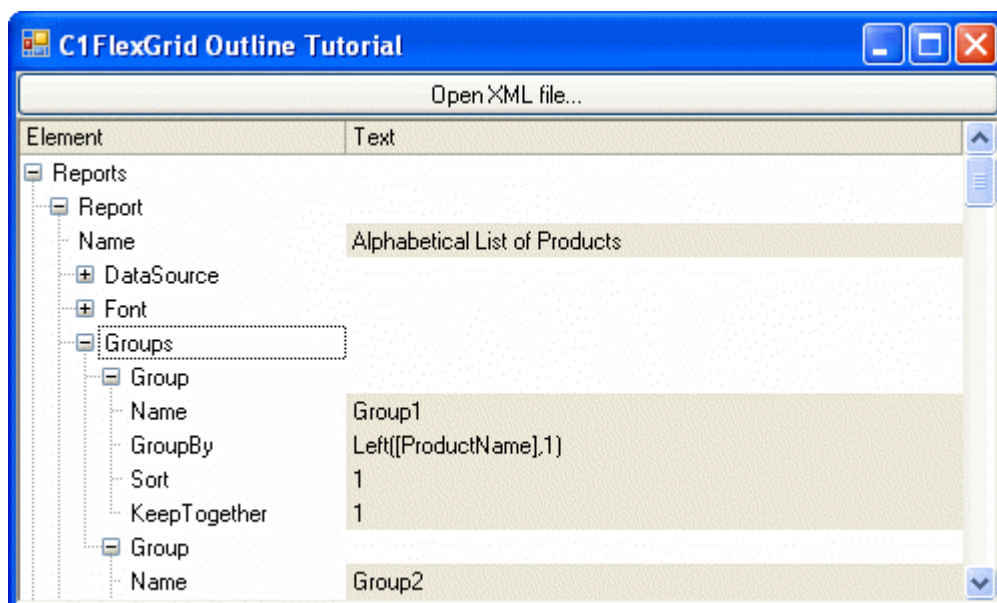
Next, the routine sets the node name and checks whether the node has exactly one child. In this case, the node is interpreted as a data node, and the node's **InnerText** property is copied to the second column on the new row. The code also sets the style of cells containing data to the custom style "Data" created when the form was loaded.

The next block of code checks to see whether this node has a subnode called "Name". If it does, then the contents of the "Name" node are assigned to the new row's **UserData** property. This value will be used later to implement ToolTips, so users can see the node name even when it is collapsed.

Finally, if the node has children, the **GetXMLData** routine calls itself to add the child nodes to the grid as well.

## Run the program and observe the following:

The project can load XML files and display them, and the user can collapse and expand nodes by clicking on them.





The next steps add a few improvements to make the application easier to use.

## Step 3 of 5: Add Custom Mouse and Keyboard Handling

The `C1FlexGrid` provides the expanding and collapsing for you, but you may extend and customize its behavior. Every time a branch is expanded or collapsed, the control fires the `BeforeCollapse` and `AfterCollapse` events so you may take actions in response to that. Furthermore, you may use the `Collapsed` property to get and set the collapsed state of each branch in code.

In this tutorial, we will trap the **DoubleClick** and **KeyPress** events to expand and collapse nodes when the user double clicks or presses the ENTER key. Add the following **DoubleClick** and **KeyPress** events to the form:

### To write code in Visual Basic

#### Visual Basic

```
Private Sub C1FlexGrid1_DoubleClick(ByVal sender As Object, ByVal e As EventArgs)
    Handles C1FlexGrid1.DoubleClick
    If C1FlexGrid1.MouseCol = 0 Then
        ToggleNodeState()
    End If
End Sub

Private Sub C1FlexGrid1_KeyPress(ByVal sender As Object, ByVal e As
    KeyPressEventArgs) Handles C1FlexGrid1.KeyPress
    If e.KeyChar = vbCr Then
        ToggleNodeState()
    End If
End Sub

Private Sub ToggleNodeState()

    ' If the current row is not a node, no work is done.
    Dim r As Row = C1FlexGrid1.Rows(C1FlexGrid1.Row)
    If Not r.IsNode Then Exit Sub

    ' Toggle the collapsed state.
    r.Node.Collapsed = Not r.Node.Collapsed
End Sub
```

### To write code in C#

#### C#

```
private void c1FlexGrid1_DoubleClick( object sender, EventArgs e)
{
    if ( c1FlexGrid1.MouseCol == 0 )
    {
        ToggleNodeState();
    }
}

private void c1FlexGrid1_KeyPress( object sender, KeyPressEventArgs e)
```

```
{
    if ( e.KeyChar == 13 )
    {
        ToggleNodeState();
    }
}

private void ToggleNodeState()
{
    // If the current row is not a node, no work is done.
    Row r = c1FlexGrid1.Rows[c1FlexGrid1.Row];
    if (! r.IsNode ) return;

    // Toggle the collapsed state.
    r.Node.Collapsed = !r.Node.Collapsed;
}
```

## Run the program and observe the following:

The event handlers check whether the user double-clicked the first column or hit the ENTER key, then call the **ToggleNodeState** routine. **ToggleNodeState** checks whether the current row is a node row, and toggles the value of the Collapsed property if it is.

## Step 4 of 5: Allow/Prevent Editing

Recall that we set the [AllowEditing](#) property of the first column to **False**. This prevents users from editing any cells in the first column. We would also like to prevent users from entering data in the second column of node rows. To do this, add the following code to handle the [BeforeEdit](#) event:

### To write code in Visual Basic

#### Visual Basic

```
Private Sub C1FlexGrid1_BeforeEdit(ByVal sender As Object, ByVal e As
RowColEventArgs) Handles C1FlexGrid1.BeforeEdit

    ' If the current row is a node, don't edit it.
    Dim r As Row = C1FlexGrid1.Rows(C1FlexGrid1.Row)
    If r.IsNode Then e.Cancel = True
End Sub
```

### To write code in C#

#### C#

```
private void c1FlexGrid1_BeforeEdit( object sender, RowColEventArgs e)
{
    // If the current row is a node, don't edit it.
    Row r = c1FlexGrid1.Rows[c1FlexGrid1.Row];
    if (r.IsNode ) e.Cancel = true;
}
```

## Step 5 of 5: Implement ToolTips

To conclude this tutorial, we will add ToolTips to the outline. The ToolTips will display the text that was stored in each row's `UserData` property by the `GetXMLData` routine described above. The ToolTips will show the contents of the "Name" node when the user moves the mouse over its parent node. This is useful when the parent node is collapsed and the "Name" node is not visible.

1. Add a **ToolTip** control to the form.
2. Add the following code to handle the grid's **MouseMove** event:

### To write code in Visual Basic

#### Visual Basic

```
Private Sub C1FlexGrid1_MouseMove(ByVal sender As Object, ByVal e As
MouseEventArgs) Handles C1FlexGrid1.MouseMove

    ' Check the ToolTip for this cell.
    Dim tip As String
    If C1FlexGrid1.MouseCol = 0 And C1FlexGrid1.MouseRow > 0 Then
        tip = C1FlexGrid1.Rows(C1FlexGrid1.MouseRow).UserData

        ' Set it if it is different from the current ToolTip.
        If tip <> ToolTip1.GetToolTip(C1FlexGrid1) Then
            ToolTip1.SetToolTip(C1FlexGrid1, tip)
        End If
    End If
End Sub
```

### To write code in C#

#### C#

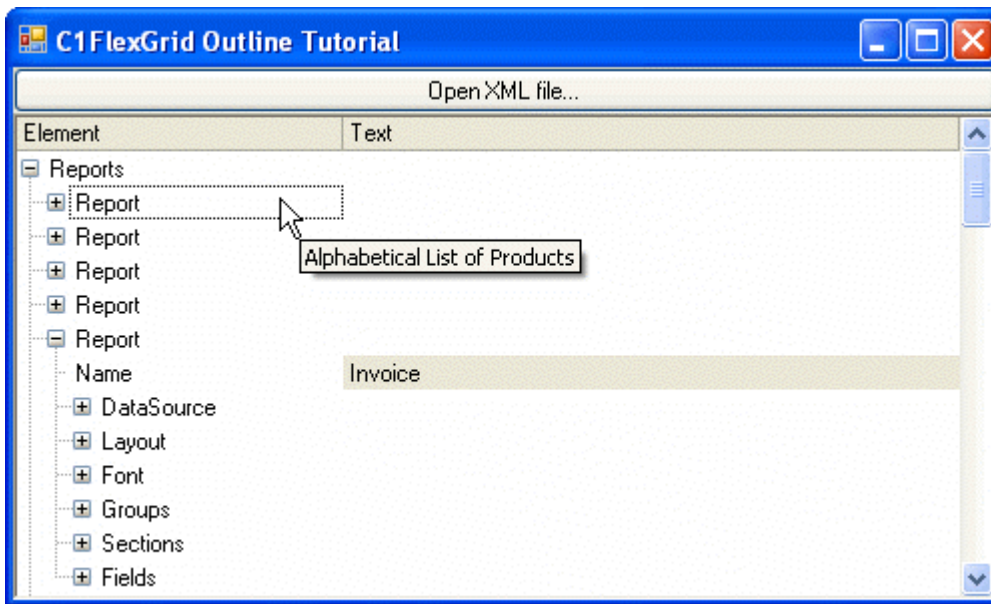
```
private void c1FlexGrid1_MouseMove( object sender, MouseEventArgs e)
{
    // Check the ToolTip for this cell.
    string tip;
    if ( c1FlexGrid1.MouseCol == 0 && c1FlexGrid1.MouseRow > 0 )
    {
        tip = (string)c1FlexGrid1.Rows[c1FlexGrid1.MouseRow].UserData;

        // Set it if it is different from the current ToolTip.
        if ( tip != toolTip1.GetToolTip(c1FlexGrid1) )
        {
            toolTip1.SetToolTip(c1FlexGrid1, tip);
        }
    }
}
```

## Run the program and observe the following:

The code starts by checking the cell under the mouse using the `MouseRow` and `MouseCol` properties. If the mouse is

over the first column on a row that contains text for the ToolTip, it retrieves the text. Otherwise, the ToolTip text is set to **Nothing**.



Then the routine compares the new and current ToolTip text, and updates the text if necessary, by calling the **SetToolTip** method on the ToolTip control.

This concludes this tutorial. You can extend this project in many ways, including saving edits back into the XML document, adding, deleting, and moving nodes, using different styles for different types of data, and so on.

## Data Analysis Tutorial

This tutorial combines some of the most useful features in the [C1FlexGrid](#) control to provide a dynamic view of a data table. The application starts with a simple data-bound grid containing sales data (from the NorthWind database), then adds the following features:

- Dynamic layout (column order)
- Automatic sorting
- Cell merging
- Automatic subtotals
- Outlining

The picture below shows what the final application looks like. The user can drag the first three columns to group the data by salesperson, country, and product name. When a column is dragged, the totals are automatically recalculated and the outline tree is rebuilt.

CategoryName	ShipCountry	LastName	ProductName	OrderDate	Sale Amount
Total for Beverages					\$309,582.25
Total for Argentina					\$1,798.00
Total for Callahan					\$514.00
Bevera...	Argentina	Callahan	Steeleye Stout	5/30/1995	\$54.00
Bevera...	Argentina	Callahan	Lakkalkiöiri	4/29/1996	\$180.00
Bevera...	Argentina	Callahan	Laughing Lumberjack Lager	5/28/1996	\$280.00
Total for Dodsworth					\$527.00
Bevera...	Argentina	Dodsworth	Côte de Blaye	2/13/1996	\$527.00
Total for Fuller					\$477.00
Bevera...	Argentina	Fuller	Rhönbräu Klosterbier	2/7/1996	\$155.00
Bevera...	Argentina	Fuller	Ipoh Coffee	2/7/1996	\$322.00
Total for King					\$280.00
Bevera...	Argentina	King	Sasquatch Ale	4/9/1996	\$280.00
Total for Austria					\$30,500.25
Total for Callahan					\$360.00
Bevera...	Austria	Callahan	Guaraní Fantástica	4/25/1996	\$360.00
Total for Davolio					\$6,409.25
Bevera...	Austria	Davolio	Chang	8/17/1994	\$950.00
Bevera...	Austria	Davolio	Rhönbräu Klosterbier	1/11/1996	\$54.25
Bevera...	Austria	Davolio	Côte de Blaye	12/12/1994	\$5,270.00

## Step 1 of 4: Create the C1FlexGrid Control for the Data Analysis Tutorial

Start a new project and add a **C1FlexGrid** control to the form by clicking the **C1FlexGrid** icon on the toolbox, then clicking on the form and dragging until the object has the proper size.

If you can't find the C1FlexGrid control in the Toolbox, right-click on the Toolbox and select **Choose Items**. Then, look for the C1FlexGrid control on the list of .NET components and make sure it is checked. If you can't find the grid in the component list, you may need to re-install the product.

1. In the **C1FlexGrid Tasks** menu, click **Dock** in parent container. This sets the grid's **Dock** property to **Fill** so the grid will fill the form.
2. Double-click the form caption area to open the code window. At the top of the file, add the following statements:

### To write code in Visual Basic

```
Visual Basic
Imports System.Data.OleDb
Imports System.ComponentModel
Imports C1.Win.C1FlexGrid
```

### To write code in C#

```
C#
using System.Data.OleDb;
using System.ComponentModel;
using C1.Win.C1FlexGrid;
```

This makes the objects defined in the C1FlexGrid and OleDb assemblies visible to the project and saves a lot of typing.

## Step 2 of 4: Initialize and Populate the Grid

To set up the grid and populate the grid with the sales data we want to analyze, set the layout properties and styles either in the designer or in code, and use the **GetDataSource** method to populate the grid.

1. Set up the grid layout and styles either in the designer or in code.

### In the Designer

- o In the Properties window, set the following properties:

Property	Setting
AllowEditing	False
AllowSorting	None
AllowMerging	Nodes
ExtendLastCol	True
SelectionMode	Cell
Tree.Style	Simple
Tree.Column	1

- o Open the **C1FlexGrid Style Editor** by selecting **Styles** from the **C1FlexGrid Tasks** menu. For more information on accessing the **C1FlexGrid Style Editor**, see [Accessing the C1FlexGrid Style Editor](#).
- o Select **Normal** from the list of **Built-In Styles**.
- o Set the **Border.Style** property to **None** and the **Trimming** property to **EllipsisCharacter**.
- o Select **Subtotal0** from the list of **Built-In Styles**.
- o Set the **BackColor** property to **Gold**, and the **ForeColor** property to **Black**.
- o Set the properties for **Subtotal1** and **Subtotal2** to the following:

Subtotal1	
BackColor	Khaki
ForeColor	Black
Subtotal2	
BackColor	LightGoldenrodYellow
ForeColor	Black

- o Click **OK** to close the editor.

### In Code

Add the following code to the **Form\_Load** event to set up the grid layout and styles:

#### To write code in Visual Basic

##### Visual Basic

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As EventArgs)
    Handles MyBase.Load

    ' Set up the grid layout/behavior.
    C1FlexGrid1.AllowEditing = False
    C1FlexGrid1.AllowSorting = AllowSortingEnum.None
    C1FlexGrid1.AllowMerging = AllowMergingEnum.Nodes
```

```

C1FlexGrid1.SelectionMode = SelectionModeEnum.Cell
C1FlexGrid1.ExtendLastCol = True
C1FlexGrid1.Cols(0).Width = C1FlexGrid1.Cols.DefaultSize / 4
C1FlexGrid1.Tree.Style = TreeStyleFlags.Simple
C1FlexGrid1.Tree.Column = 1

' Set up grid styles.
C1FlexGrid1.Styles.Normal.Border.Style = BorderStyleEnum.None
C1FlexGrid1.Styles.Normal.Trimming = StringTrimming.EllipsisCharacter

Dim s As CellStyle = C1FlexGrid1.Styles(CellStyleEnum.GrandTotal)
s.BackColor = Color.Black
s.ForeColor = Color.White
s = C1FlexGrid1.Styles(CellStyleEnum.Subtotal0)
s.BackColor = Color.Gold
s.ForeColor = Color.Black
s = C1FlexGrid1.Styles(CellStyleEnum.Subtotal1)
s.BackColor = Color.Khaki
s.ForeColor = Color.Black
s = C1FlexGrid1.Styles(CellStyleEnum.Subtotal2)
s.BackColor = Color.LightGoldenrodYellow
s.ForeColor = Color.Black
End Sub

```

**To write code in C#**

```

C#
private void Form1_Load( System.Object sender, EventArgs e)
{
    // Set up the grid layout/behavior.
    c1FlexGrid1.AllowEditing = false;
    c1FlexGrid1.AllowSorting = AllowSortingEnum.None;
    c1FlexGrid1.AllowMerging = AllowMergingEnum.Nodes;
    c1FlexGrid1.SelectionMode = SelectionModeEnum.Cell;
    c1FlexGrid1.ExtendLastCol = true;
    c1FlexGrid1.Cols[0].Width = c1FlexGrid1.Cols.DefaultSize / 4;
    c1FlexGrid1.Tree.Style = TreeStyleFlags.Simple;
    c1FlexGrid1.Tree.Column = 1;

    // Set up grid styles.
    c1FlexGrid1.Styles.Normal.Border.Style = BorderStyleEnum.None;
    c1FlexGrid1.Styles.Normal.Trimming = StringTrimming.EllipsisCharacter;

    CellStyle s = c1FlexGrid1.Styles[CellStyleEnum.GrandTotal];
    s.BackColor = Color.Black;
    s.ForeColor = Color.White;
    s = c1FlexGrid1.Styles[CellStyleEnum.Subtotal0];
    s.BackColor = Color.Gold;
    s.ForeColor = Color.Black;
    s = c1FlexGrid1.Styles[CellStyleEnum.Subtotal1];
    s.BackColor = Color.Khaki;

```

```
s.ForeColor = Color.Black;
s = c1FlexGrid1.Styles[CellStyleEnum.Subtotal2];
s.BackColor = Color.LightGoldenrodYellow;
s.ForeColor = Color.Black;
}
```

The routine starts by setting up the grid layout and some styles.

2. Bind C1FlexGrid to a data source by adding the following code to the **Form\_Load** event:

#### To write code in Visual Basic

##### Visual Basic

```
' Bind C1FlexGrid to the data source.
C1FlexGrid1.DataSource = GetDataSource()
```

#### To write code in C#

##### C#

```
// Bind C1FlexGrid to the data source.
c1FlexGrid1.DataSource = GetDataSource();
```

The routine binds it to a data source created by the **GetDataSource** method, listed below.

3. Lock the last three columns in place by setting their [AllowDragging](#) property to **False**. This is done to prevent the user from grouping the data in these columns (the values in these columns are distinct for each row). This property can be set either in the designer or at in code.

#### In the Designer

- o Select Column 4 in the grid. This will open the **Column Tasks** menu for Column 4.
- o Uncheck the **Allow Dragging** check box.
- o Repeat for Column 5 and Column 6.

Alternatively, the [AllowDragging](#) property can also be set using the **C1FlexGrid Column Editor**:

- o Open the **C1FlexGrid Column Editor** by selecting **Designer** in the **C1FlexGrid Tasks** menu. For details on how to access the **C1FlexGrid Column Editor**, see [Accessing the C1FlexGrid Column Editor](#).
- o Select Column 4 in the right pane.
- o In the left pane, set the **AllowDragging** property to **False**.
- o Set the **AllowDragging** property to **False** for Column 5 and Column 6.
- o Do not close the editor.

#### In Code

Add the following code to the **Form\_Load** event:

#### To write code in Visual Basic

##### Visual Basic

```
' Prevent the user from dragging last three columns.
C1FlexGrid1.Cols(4).AllowDragging = False
C1FlexGrid1.Cols(5).AllowDragging = False
C1FlexGrid1.Cols(6).AllowDragging = False
```



**To write code in C#**

```
C#  
  
// Prevent the user from dragging last three columns.  
c1FlexGrid1.Cols[4].AllowDragging = false;  
c1FlexGrid1.Cols[5].AllowDragging = false;  
c1FlexGrid1.Cols[6].AllowDragging = false;
```

4. Set the **Format** property of the *Sales Amount* column so that the amounts are displayed as currency values. This can be done either in the designer or in code.

**In the Designer**

- o Select Column 6 in the grid.
- o Click the **ellipsis** button in the **Format String** box to open the **Format String** dialog box.
- o Under Format type select **Currency**.
- o Click **OK** to close the **Format String** dialog box.

Alternatively, the Format property can also be set using the **C1FlexGrid Column Editor**:

- o In the **C1FlexGrid Column Editor**, select Column 6 in the right pane.
- o In the left pane, click the **ellipsis** button next to the **Format** property to open the **Format String** dialog box.
- o Under Format type select Currency.
- o Click **OK** to close the **Format String** dialog box.
- o Click **OK** to close the editor.

**In Code****To write code in Visual Basic**

```
Visual Basic  
  
' Display currency values in the Sales Amount column.  
C1FlexGrid1.Cols(6).Format = "c"
```

**To write code in C#**

```
C#  
  
// Display currency values in the Sales Amount column.  
c1FlexGrid1.Cols[6].Format = "c";
```

5. The **GetDataSource** method creates the data table that is displayed by the grid. The routine is very basic, except for the SQL statement that retrieves the data. Most people don't write these SQL statements manually, but use visual designers such as the one in Visual Studio or Microsoft Access to do that.

Add the following code to the form. Note that you may have to change the connection string slightly, because it has a reference to the NorthWind database and that file might be in a different folder in your system:

**To write code in Visual Basic**

```
Visual Basic  
  
Private Function GetDataSource() As DataTable  
  
    ' Set up the connection string.  
    Dim conn As String = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
```

```

    "Data Source=C:\Users\\Documents\ComponentOne
Samples\Common\C1NWind.mdb"

    ' Set up the SQL statement.
Dim rs As String = _
    "SELECT Employees.LastName,Orders.ShipCountry," & _
    "Categories.CategoryName,Products.ProductName,Orders.OrderDate," & _
    "[Quantity]*[Products].[UnitPrice] AS [Sale Amount] " & _
    "FROM (Categories INNER JOIN Products " & _
    "ON Categories.CategoryID = Products.CategoryID) " & _
    "INNER JOIN ((Employees INNER JOIN Orders " & _
    "ON Employees.EmployeeID = Orders.EmployeeID) " & _
    "INNER JOIN [Order Details] " & _
    "ON Orders.OrderID = [Order Details].OrderID) " & _
    "ON Products.ProductID = [Order Details].ProductID " & _
    "ORDER BY Employees.LastName,Orders.ShipCountry," & _
    "Categories.CategoryName;"

    ' Retrieve the data into the DataSet.
Dim da As OleDbDataAdapter = New OleDbDataAdapter(rs, conn)
Dim ds As DataSet = New DataSet()
da.Fill(ds)

    ' Return the data table.
GetDataSource = ds.Tables(0)
End Function

```

**To write code in C#**

```

C#
private DataTable GetDataSource()
{
    // Set up the connection string.
    string conn = "Provider=Microsoft.Jet.OLEDB.4.0;" +
        "Data Source=C:\\Users\\Windows 8.1\\Documents\\ComponentOne
Samples\\Common\\C1NWind.mdb";

    // Set up the SQL statement.
    string rs =
        "SELECT Employees.LastName,Orders.ShipCountry," +
        "Categories.CategoryName,Products.ProductName,Orders.OrderDate," +
        "[Quantity]*[Products].[UnitPrice] AS [Sale Amount] " +
        "FROM (Categories INNER JOIN Products " +
        "ON Categories.CategoryID = Products.CategoryID) " +
        "INNER JOIN ((Employees INNER JOIN Orders " +
        "ON Employees.EmployeeID = Orders.EmployeeID) " +
        "INNER JOIN [Order Details] " +
        "ON Orders.OrderID = [Order Details].OrderID) " +
        "ON Products.ProductID = [Order Details].ProductID " +
        "ORDER BY Employees.LastName,Orders.ShipCountry," +
        "Categories.CategoryName;";
}

```

```

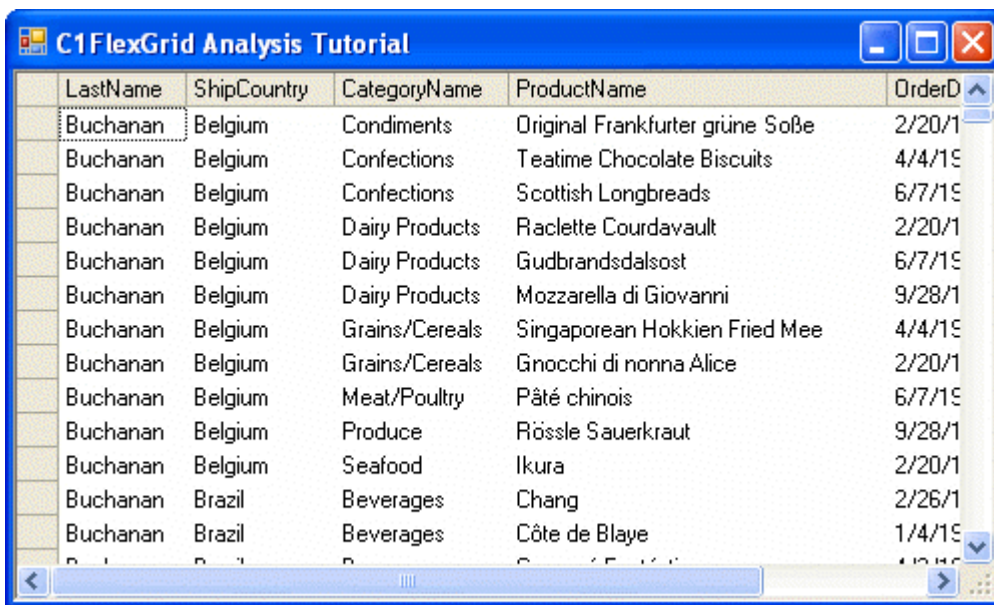
// Retrieve the data into the DataSet.
OleDbDataAdapter da = new OleDbDataAdapter(rs, conn);
DataSet ds = new DataSet();
da.Fill(ds);

// Return the data table.
return ds.Tables[0];
}

```

## Run the program and observe the following:

You will see a plain-looking grid that allows you to move columns around and browse through the data. However, the data is not structured in a clear way, and this table contains a couple of thousand records, so it is pretty difficult to get an overview of what the data means.



## Step 3 of 4: Allow Automatic Sorting

The first step in organizing the data is sorting it. Furthermore, we would like the data to be sorted automatically whenever the user reorders the columns.

After the user reorders the columns, the `C1FlexGrid` control fires the `AfterDragColumn` event. We will add an event handler to sort the data in the underlying data table. (If the grid were being used in unbound mode, we would accomplish this using the `Sort` method.)

Add the following code to the form to sort the record set and rebuild the subtotals when the user drags columns:

### To write code in Visual Basic

#### Visual Basic

```

Private Sub C1FlexGrid1_AfterDragColumn(ByVal sender As Object, ByVal e As
DragRowColEventArgs) Handles C1FlexGrid1.AfterDragColumn

```

```

' Sort the recordset when the user drags columns.
' This will cause a data refresh, removing all subtotals and
' firing the AfterDataRefresh event, which rebuilds the subtotals.
Dim sort As String = C1FlexGrid1.Cols(1).Name & ", " & _
                    C1FlexGrid1.Cols(2).Name & ", " & _
                    C1FlexGrid1.Cols(3).Name

Dim dt As DataTable = C1FlexGrid1.DataSource
dt.DefaultView.Sort = sort
End Sub

```

## To write code in C#

```

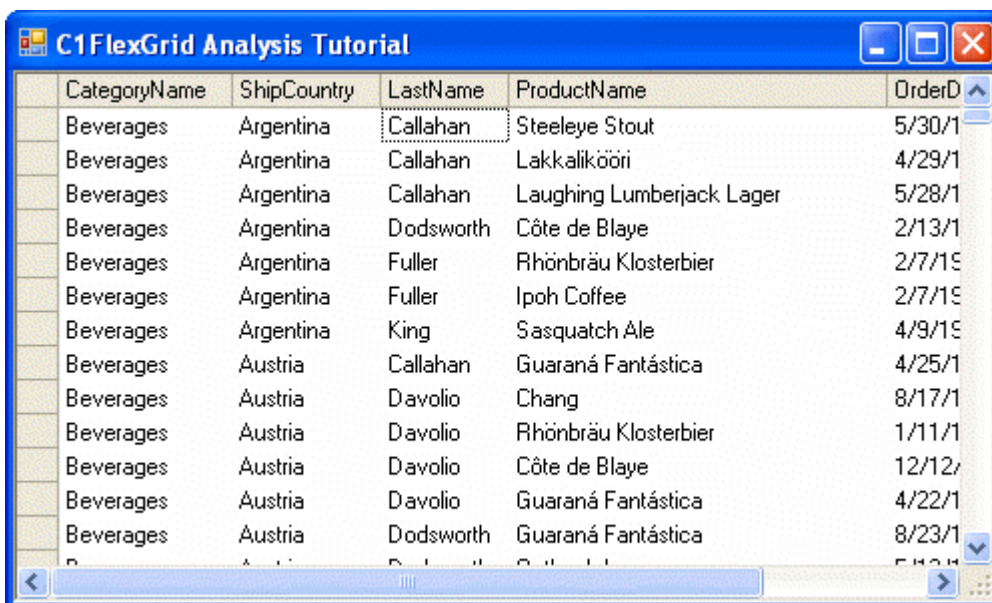
C#
private void c1FlexGrid1_AfterDragColumn( object sender, DragRowColEventArgs e)
{
    // Sort the recordset when the user drags columns.
    // This will cause a data refresh, removing all subtotals and
    // firing the AfterDataRefresh event, which rebuilds the subtotals.
    string sort = c1FlexGrid1.Cols[1].Name + ", " +
                 c1FlexGrid1.Cols[2].Name + ", " +
                 c1FlexGrid1.Cols[3].Name;

    DataTable dt = (DataTable)c1FlexGrid1.DataSource;
    dt.DefaultView.Sort = sort;
}

```

## Run the program and observe the following:

Run the project and try reordering the first three columns by dragging their headings around. Whenever you move a column, the data is automatically sorted, which makes it easier to interpret.



In the next step, we will add subtotals and an outline tree.

## Run the program and observe the following:

The code starts by getting the index of the *Sale Amount* column. In this tutorial, the index will always be the same (Column 6). Looking up the index is usually better than hardwiring it, though, because if someone added a couple of fields to the SQL statement the index would change.

The code then calls the `C1FlexGrid.Subtotal` method to group the data and insert new rows with the subtotals. The new rows are automatically configured as outline nodes (their `IsNode` property is set to **True**), so the subtotals are collapsible.

LastName	ShipCountry	CategoryName	ProductName	OrderDate	Sale Amount
Total for Buchanan					\$81,745.85
Total for Belgium					\$7,854.20
Total for Condiments					\$195.00
Buchan...	Belgium	Condiments	Original Frankfurter grüne Soße	2/20/1996	\$195.00
Total for Confections					\$443.20
Buchan...	Belgium	Confections	Teatime Chocolate Biscuits	4/4/1995	\$193.20
Buchan...	Belgium	Confections	Scottish Longbreads	6/7/1995	\$250.00
Total for Dairy Products					\$3,632.00
Buchan...	Belgium	Dairy Products	Raclette Courdevault	2/20/1996	\$2,750.00
Buchan...	Belgium	Dairy Products	Gudbrandsdalsost	6/7/1995	\$360.00
Buchan...	Belgium	Dairy Products	Mozzarella di Giovanni	9/28/1995	\$522.00
Total for Grains/Cereals					\$1,840.00
Buchan...	Belgium	Grains/Cereals	Singaporean Hokkien Fried Mee	4/4/1995	\$700.00
Buchan...	Belgium	Grains/Cereals	Gnocchi di nonna Alice	2/20/1996	\$1,140.00
Total for Meat/Poultry					\$336.00
Buchan...	Belgium	Meat/Poultry	Pâté chinois	6/7/1995	\$336.00
Total for Produce					\$912.00
Buchan...	Belgium	Produce	Rössle Sauerkraut	9/28/1995	\$912.00
Total for Seafood					\$496.00
Buchan...	Belgium	Seafood	Ikura	2/20/1996	\$496.00

Try dragging columns around. You can easily see the totals by country, product category, or salesperson. You can also expand tree nodes to drill down into the data if you want to see more detail.

CategoryName	ShipCountry	LastName	ProductName	OrderDate	Sale Amount
Total for Beverages					\$309,582.00
Total for Argentina					\$1,798.00
Total for Callahan					\$514.00
Bevera...	Argentina	Callahan	Steeleye Stout	5/30/1995	\$54.00
Bevera...	Argentina	Callahan	Lakkaikööri	4/29/1996	\$180.00
Bevera...	Argentina	Callahan	Laughing Lumberjack Lager	5/28/1996	\$280.00
Total for Dodsworth					\$527.00
Bevera...	Argentina	Dodsworth	Côte de Blaye	2/13/1996	\$527.00
Total for Fuller					\$477.00
Bevera...	Argentina	Fuller	Rhönbräu Klosterbier	2/7/1996	\$155.00
Bevera...	Argentina	Fuller	Ipoh Coffee	2/7/1996	\$322.00
Total for King					\$280.00
Bevera...	Argentina	King	Sasquatch Ale	4/9/1996	\$280.00
Total for Austria					\$30,500.00
Total for Callahan					\$360.00
Total for Davolio					\$6,409.00
Total for Dodsworth					\$576.00
Total for Fuller					\$1,380.00
Total for King					\$14,700.00

Note also that the grid is editable, changing some values in the *Sale Amount* column will cause the *AfterDataRefresh* event to fire again, and the totals will be automatically updated.

This concludes this tutorial.

## FlexGrid for WinForms Task-Based Help

The task-based help assumes that you are familiar with programming in Visual Studio .NET, and know how to use bound and unbound controls in general. If you are a novice to the **FlexGrid for WinForms** product, please see the [FlexGrid for WinForms Tutorials](#) first.

Each topic provides a solution for specific tasks using the **FlexGrid for WinForms** product. By following the steps outlined in the help, you will be able to create projects demonstrating a variety of **C1FlexGrid** features.

Each task-based help topic also assumes that you have created a new .NET project.

## Accessing the C1FlexGrid Editors

The **C1FlexGrid** editors can be accessed through the **C1FlexGrid Tasks** menu, the context menu, or the Properties window. There are two **C1FlexGrid** editors, the **C1FlexGrid Column Editor** and the **C1FlexGrid Style Editor**, which allow you to control the layout and appearance of **C1FlexGrid** at design time. To reorder columns, adjust column widths, set column properties, and insert or remove columns, use the **C1FlexGrid Column Editor**. To modify existing styles and add custom styles, which may later be assigned to cells, rows, and columns, use the **C1FlexGrid Style Editor**.

## Accessing the C1FlexGrid Column Editor

To access the **C1FlexGrid Column Editor**, use the **C1FlexGrid Tasks** menu, the context menu, or the Properties window. For more information on editing columns with the **C1FlexGrid Column Editor**, see [C1FlexGrid Column Editor](#).

## C1FlexGrid Tasks Menu

Click the smart tag (  ) in the upper right corner of **C1FlexGrid** to open the **C1FlexGrid Tasks** menu, and select **Designer**.

## Context Menu

Right-click on the form and select **Designer** from the context menu.

## Properties Window

In the Properties window, click the **ellipsis** button next to the **Cols** property.

## Accessing the C1FlexGrid Style Editor

To access the **C1FlexGrid Style Editor**, use the **C1FlexGrid Tasks** menu, the context menu, or the Properties window. For more information on customizing cell appearance with the **C1FlexGrid Style Editor**, see [C1FlexGrid Style Editor](#).

## C1FlexGrid Tasks Menu

Click the smart tag (  ) in the upper right corner of **C1FlexGrid** to open the **C1FlexGrid Tasks** menu, and select **Styles**.

## Context Menu

Right-click on the form and select **Styles** from the context menu.

## Properties Window

In the Properties window, click the **ellipsis** button next to the **Styles** property.

## Adding Pictures and Text to a Cell

To add pictures and text to a cell, use the [SetData](#) and [SetCellImage](#) methods. Add the following code to the **Form\_Load** event:

1. Resize the cells to fit the image using the [Height](#) and [Width](#) properties.

### To write code in Visual Basic

```
Visual Basic
Me.C1FlexGrid1.Rows(1).Height = 90
Me.C1FlexGrid1.Cols(1).Width = 150
```

### To write code in C#

```
C#
this.c1FlexGrid1.Rows[1].Height = 90;
this.c1FlexGrid1.Cols[1].Width = 150;
```

2. Add the [SetCellImage](#) method to add the image.

### To write code in Visual Basic

```
Visual Basic
Me.C1FlexGrid1.SetCellImage(1, 1, Image.FromFile("c:\c1logo.bmp"))
```

### To write code in C#

```
C#
this.c1FlexGrid1.SetCellImage(1, 1, Image.FromFile(@"c:\c1logo.bmp"));
```

3. Add the [SetData](#) method to add the text:

### To write code in Visual Basic

```
Visual Basic
Me.C1FlexGrid1.SetData(1, 1, "ComponentOne")
```

### To write code in C#

```
C#
this.c1FlexGrid1.SetData(1, 1, "ComponentOne");
```



4. Set the alignment of the image to **CenterTop** and the text to **CenterBottom**:

**To write code in Visual Basic**

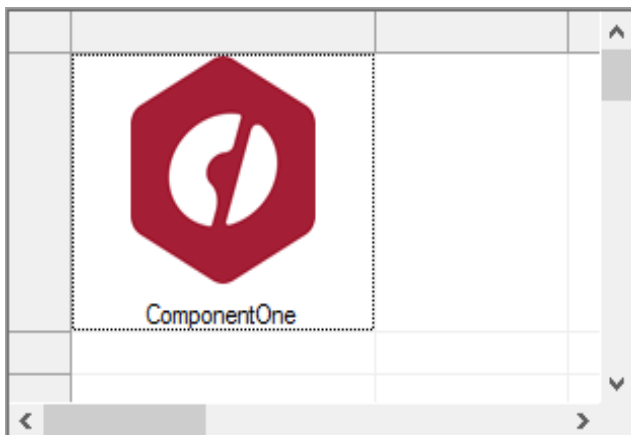
```
Visual Basic
Me.C1FlexGrid1.Styles.Normal.ImageAlign =
C1.Win.C1FlexGrid.ImageAlignEnum.CenterTop
Me.C1FlexGrid1.Styles.Normal.TextAlign =
C1.Win.C1FlexGrid.TextAlignEnum.CenterBottom
```

**To write code in C#**

```
C#
this.c1FlexGrid1.Styles.Normal.ImageAlign =
C1.Win.C1FlexGrid.ImageAlignEnum.CenterTop;
this.c1FlexGrid1.Styles.Normal.TextAlign =
C1.Win.C1FlexGrid.TextAlignEnum.CenterBottom;
```

## This topic illustrates the following:

Your grid will look like the following with an image and text in the same cell.



**Note:** To set the text on top of the picture, change the alignment of the text to **CenterTop**, and the alignment of the image to **CenterBottom**.

## Adding Row Numbers in a Fixed Column

To add row numbers in a fixed column, like in Microsoft Excel, use the [OwnerDrawCell](#) event to draw the numbers, aligned left, in a fixed column.

1. Add the following code to the **Form\_Load** event to fire the `OwnerDrawCell` event:

**To write code in Visual Basic**

```
Visual Basic
Me.C1FlexGrid1.DrawMode = C1.Win.C1FlexGrid.DrawModeEnum.OwnerDraw
```

**To write code in C#**

C#

```
this.c1FlexGrid1.DrawMode = C1.Win.C1FlexGrid.DrawModeEnum.OwnerDraw;
```

2. Add the OwnerDrawCell event:

**To write code in Visual Basic**

Visual Basic

```
Private Sub C1FlexGrid1_OwnerDrawCell(ByVal sender As Object, ByVal e As  
C1.Win.C1FlexGrid.OwnerDrawCellEventArgs) Handles C1FlexGrid1.OwnerDrawCell  
    If e.Row >= Me.C1FlexGrid1.Rows.Fixed And e.Col = Me.C1FlexGrid1.Cols.Fixed  
    - 1 Then  
        Dim rowNumber As Integer = e.Row - Me.C1FlexGrid1.Rows.Fixed + 1  
        e.Text = rowNumber.ToString()  
    End If  
End Sub
```

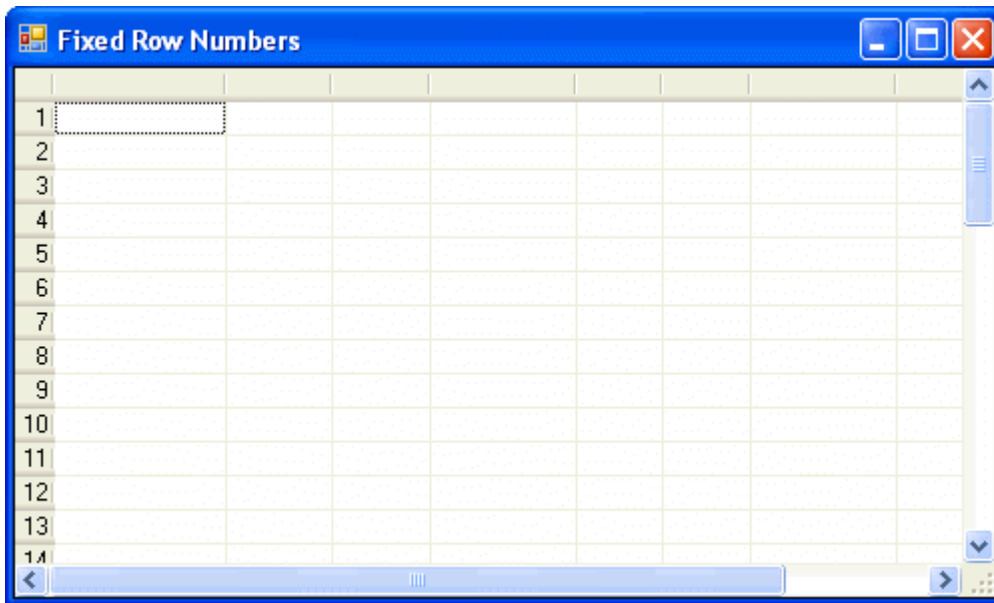
**To write code in C#**

C#

```
private void c1FlexGrid1_OwnerDrawCell(object sender,  
C1.Win.C1FlexGrid.OwnerDrawCellEventArgs e)  
{  
    if ((e.Row >= this.c1FlexGrid1.Rows.Fixed) & (e.Col ==  
(this.c1FlexGrid1.Cols.Fixed - 1)))  
    {  
        e.Text = ((e.Row - this.c1FlexGrid1.Rows.Fixed) + 1).ToString();  
    }  
}
```

**This topic illustrates the following:**

Row numbers appear in the first column, which is fixed, and are left-aligned like in Microsoft Excel.



## Adding Three-Dimensional Text to a Header Row

To add three-dimensional text to the table's header row, set the `TextEffect` property to **Raised** for text with a shadow offset by one pixel to the right and below the text or **Inset** for text with a shadow offset by one pixel to the left and above the text. Setting the `TextEffect` property to **Flat** gives text no effect.

1. Create a new style called **3DText**.

### In the Designer

- o Open the **C1FlexGrid Style Editor**. For details on how to access the **C1FlexGrid Style Editor**, see [Accessing the C1FlexGrid Style Editor](#).
- o Click **Add** to create a new style.
- o Double-click **CustomStyle1**, rename it **3DText**, and press ENTER when finished.
- o Do not exit the **C1FlexGrid Style Editor**.

### In Code

Add the following code to the **Form\_Load** event:

#### To write code in Visual Basic

```
Visual Basic
Dim tdt As C1.Win.C1FlexGrid.CellStyle
```

#### To write code in C#

```
C#
C1.Win.C1FlexGrid.CellStyle tdt = this.c1FlexGrid1.Styles.Add("3Dtext");
```

2. Set the `TextEffect` property to **Raised**.

### In the Designer

Locate the `TextEffect` property in the right pane and set it to **Raised**.

### In Code

Add the following code to the **Form\_Load** event:

**To write code in Visual Basic**

Visual Basic

```
tdt.TextEffect = C1.Win.C1FlexGrid.TextEffectEnum.Raised
```

**To write code in C#**

C#

```
tdt.TextEffect = C1.Win.C1FlexGrid.TextEffectEnum.Raised;
```

3. Apply the style to the header row by adding the following code to the **Form\_Load** event:

**To write code in Visual Basic**

Visual Basic

```
Me.C1FlexGrid1.Rows(0).Style = Me.C1FlexGrid1.Styles("3DText")
```

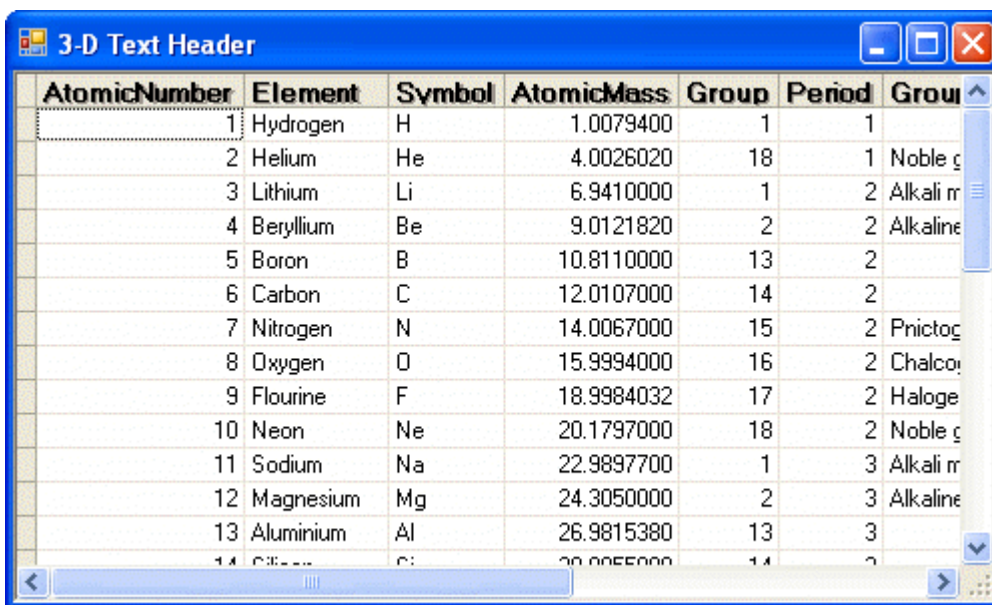
**To write code in C#**

C#

```
this.c1FlexGrid1.Rows[0].Style = this.c1FlexGrid1.Styles["3DText"];
```

## This topic illustrates the following:

The table should have a header with raised text similar to the following image.



## Adding Three-Dimensional Text to a Header Row Using Built-In Styles

The **TextEffect** property can be used to add a three-dimensional text effect to **C1FlexGrid**'s built-in styles. To add a header with three-dimensional text using styles, set the **TextEffect** property within a style to **Raised** for text with a shadow offset by one pixel to the right and below the text or **Inset** for text with a shadow offset by one pixel to the left and above the text. Setting the **TextEffect** property to **Flat** gives text no effect. This property can be set either in the designer or in code.

### In the Designer

1. Open the **C1FlexGrid Style Editor**. For details on how to access the **C1FlexGrid Style Editor**, see [Accessing the C1FlexGrid Style Editor](#).
2. Select **Fixed** under **Built-In Styles**.
3. In the right column, locate the **TextEffect** property and set it to **Raised**.
4. Click **OK** to close the editor.

### In Code

Add the following code to the **Form\_Load** event to set the **TextEffect** property in the Fixed style:

#### To write code in Visual Basic

Visual Basic

```
Me.C1FlexGrid1.Styles("Fixed").TextEffect = C1.Win.C1FlexGrid.TextEffectEnum.Raised
```

#### To write code in C#

C#

```
this.c1FlexGrid1.Styles["Fixed"].TextEffect =  
C1.Win.C1FlexGrid.TextEffectEnum.Raised;
```

### This topic illustrates the following:

By setting the **TextEffect** property in the Fixed style, only fixed cells in the table will have a raised three-dimensional effect. The **TextEffect** property can also be set in the all of the built-in styles. See [CellStyleEnum Enumeration](#) for a list and descriptions of the built-in styles.

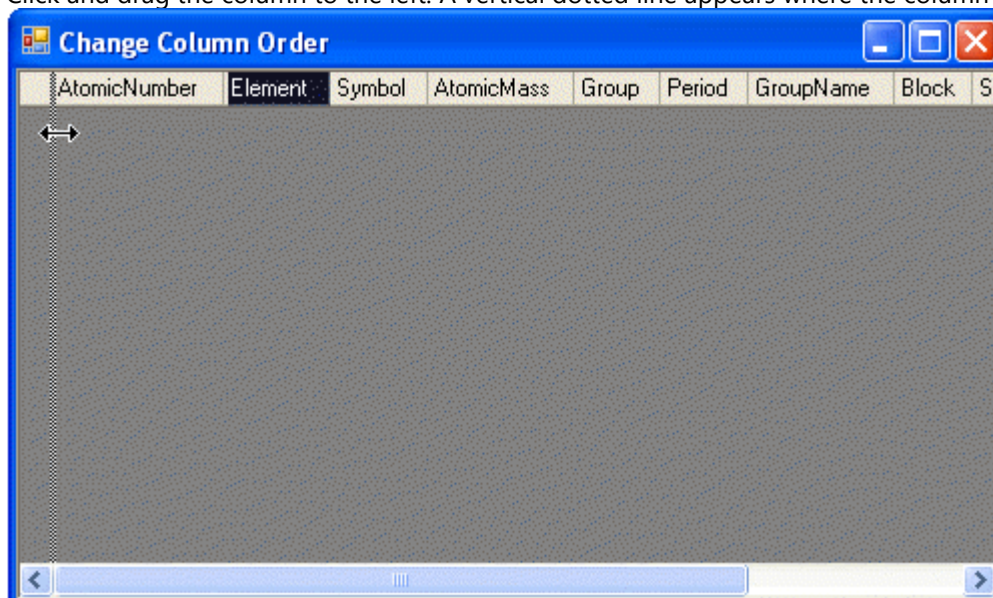
AtomicNumber	Element	Symbol	AtomicMass	Group	Period	Group
1	Hydrogen	H	1.0079400	1	1	
2	Helium	He	4.0026020	18	1	Noble g
3	Lithium	Li	6.9410000	1	2	Alkali m
4	Beryllium	Be	9.0121820	2	2	Alkaline
5	Boron	B	10.8110000	13	2	
6	Carbon	C	12.0107000	14	2	
7	Nitrogen	N	14.0067000	15	2	Prictog
8	Oxygen	O	15.9994000	16	2	Chalcoy
9	Flourine	F	18.9984032	17	2	Haloge
10	Neon	Ne	20.1797000	18	2	Noble g
11	Sodium	Na	22.9897700	1	3	Alkali m
12	Magnesium	Mg	24.3050000	2	3	Alkaline
13	Aluminium	Al	26.9815380	13	3	
14	Silicon	Si	28.0855000	14	3	

## Changing the Column Order in the Grid

To change the column order in the grid, drag the column to the new position in the grid or use the **C1FlexGrid Column Editor**, or set the [MoveRange](#) method in code.

### In the Designer

1. In the grid, select the column you would like to move. In this example, the *Element* column will be moved.
2. Click and drag the column to the left. A vertical dotted line appears where the column can be dropped.



3. Drop the *Element* column before the *AtomicNumber* column.

Alternatively, columns can be reordered in the grid using the **C1FlexGrid Column Editor**:

1. Open the **C1FlexGrid Column Editor**. For details on how to access the **C1FlexGrid Column Editor**, see [Accessing the C1FlexGrid Column Editor](#).

2. In the designer, select the column you would like to move. In this example, the *Element* column will be moved.
3. Click and drag the column to the left. A vertical dotted line appears where the column can be dropped.
4. Drop the *Element* column before the *AtomicNumber* column.
5. Click **OK** to close the editor.

## In Code

Add the following code to the **Form\_Load** event to move Column 2 (the *Element* column in this example) to the Column 1 position:

### To write code in Visual Basic

Visual Basic

```
Me.C1FlexGrid1.Cols.MoveRange(2, 1, 1)
```

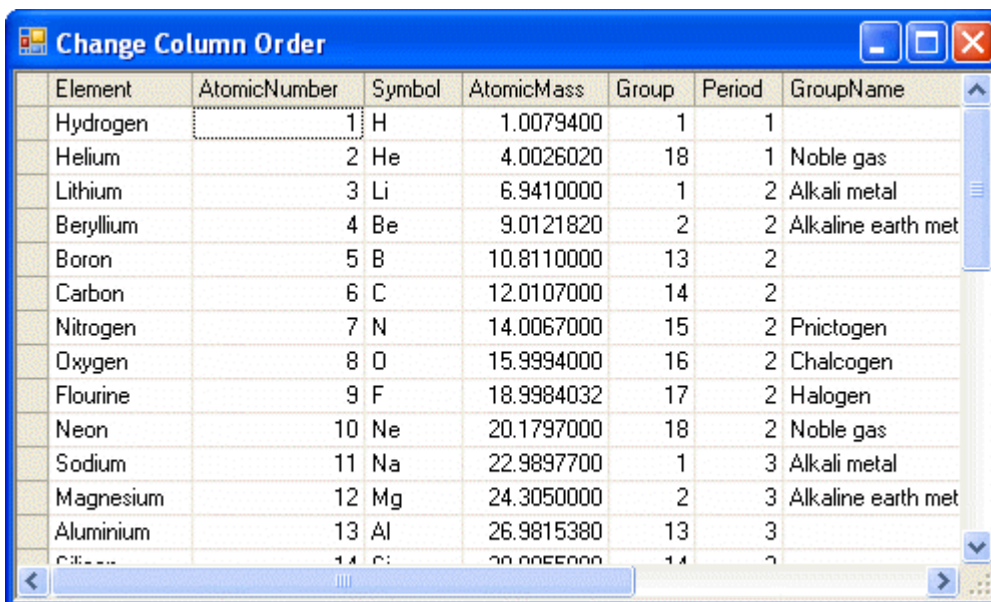
### To write code in C#

C#

```
this.c1FlexGrid1.Cols.MoveRange(2, 1, 1);
```

## This topic illustrates the following:

The *Element* column now appears before the *AtomicNumber* column in the grid.



Element	AtomicNumber	Symbol	AtomicMass	Group	Period	GroupName
Hydrogen	1	H	1.0079400	1	1	
Helium	2	He	4.0026020	18	1	Noble gas
Lithium	3	Li	6.9410000	1	2	Alkali metal
Beryllium	4	Be	9.0121820	2	2	Alkaline earth met
Boron	5	B	10.8110000	13	2	
Carbon	6	C	12.0107000	14	2	
Nitrogen	7	N	14.0067000	15	2	Prictogen
Oxygen	8	O	15.9994000	16	2	Chalcogen
Flourine	9	F	18.9984032	17	2	Halogen
Neon	10	Ne	20.1797000	18	2	Noble gas
Sodium	11	Na	22.9897700	1	3	Alkali metal
Magnesium	12	Mg	24.3050000	2	3	Alkaline earth met
Aluminium	13	Al	26.9815380	13	3	
Silicon	14	Si	28.0855000	14	3	

## Filtering by Value

To use the **ValueFilter**, follow these steps:

1. Select the grid and click the smart tag to open the **C1FlexGrid Tasks** menu.
2. Select the **Enable Column Filtering** check box.
3. Click the **Designer** link. The **C1FlexGrid Column Editor** appears.

- Click the drop-down arrow next to the [AllowFiltering](#) property and select **ByValue**.

## In Code

Add the following code to the **Form\_Load** event:

### To write code in Visual Basic

Visual Basic

```
Me.C1FlexGrid1.AllowFiltering = True
Me.C1FlexGrid1.Cols(1).AllowFiltering = AllowFiltering.ByValue
```

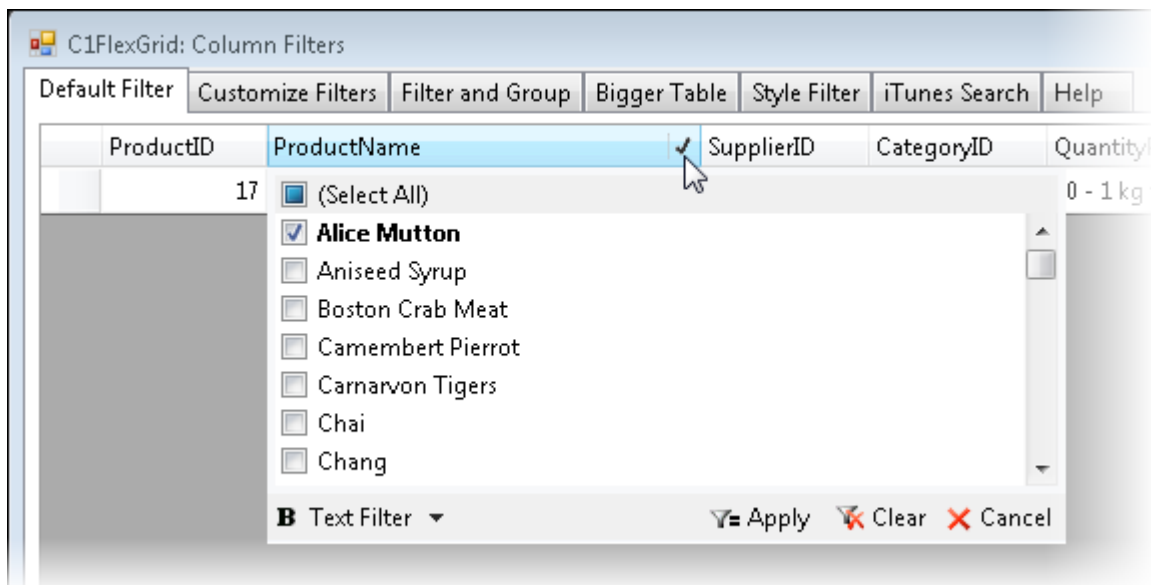
### To write code in C#

C#

```
this.c1FlexGrid1.AllowFiltering = true;
this.c1FlexGrid1.Cols[1].AllowFiltering = AllowFiltering.ByValue
```

## This topic illustrates the following:

In this example, the second column is sorted by value:



## Filtering by Condition

To use the **ConditionFilter**, follow these steps:

- Select the grid and click the smart tag to open the **C1FlexGrid Tasks** menu.
- Select the **Enable Column Filtering** check box.
- Click the **Designer** link. The **C1FlexGrid Column Editor** appears.
- Click the drop-down arrow next to the [AllowFiltering](#) property and select **ByCondition**.



## In Code

Add the following code to the **Form\_Load** event:

### To write code in Visual Basic

Visual Basic

```
Me.C1FlexGrid1.AllowFiltering = True
Me.C1FlexGrid1.Cols(1).AllowFiltering = AllowFiltering.ByCondition
```

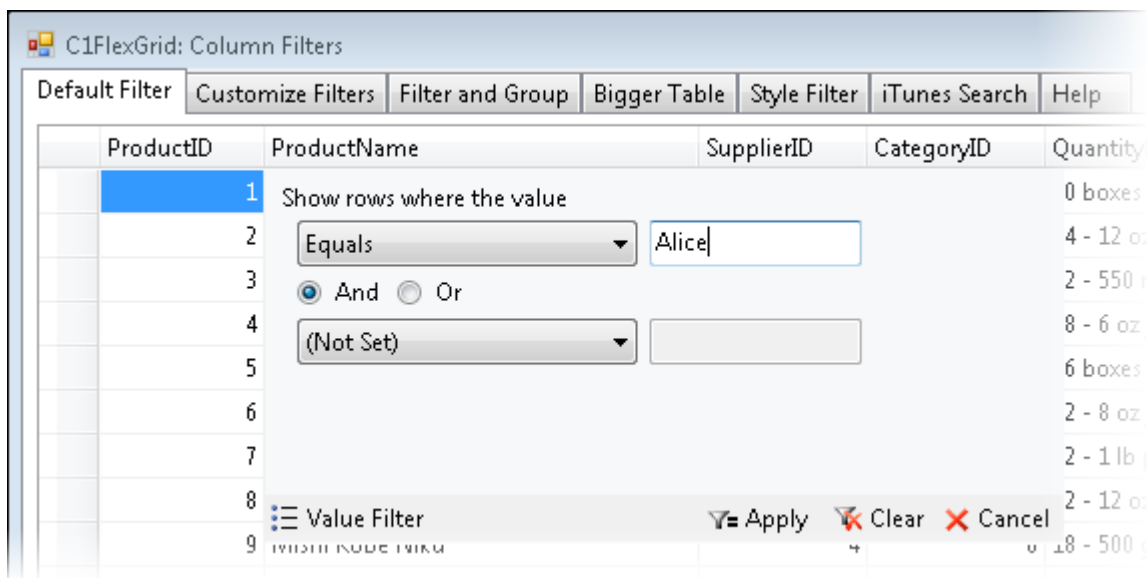
### To write code in C#

C#

```
this.c1FlexGrid1.AllowFiltering = true;
this.c1FlexGrid1.Cols[1].AllowFiltering = AllowFiltering.ByCondition
```

## This topic illustrates the following:

In this example, the second column is sorted by condition:



## Changing the Filter Language

To change the language used in the column filter editor, you can use the [Language](#) property.

1. Right-click your grid and select **Properties** to view the Visual Studio Properties window.
2. Set the [AllowFiltering](#) property to **True**.
3. Click the drop-down arrow next to the Language property and select a language.
4. Run the project and click the drop-down arrow on one of the column headers to open the column filter editor. The language of the column filter editor matches the language specified in the Language property.

## In Code

Add the following code to the **Form\_Load** event:

### To write code in Visual Basic

Visual Basic

```
Me.C1FlexGrid1.AllowFiltering = True  
Me.C1FlexGrid1.Language = C1.Util.Localization.Language.Danish
```

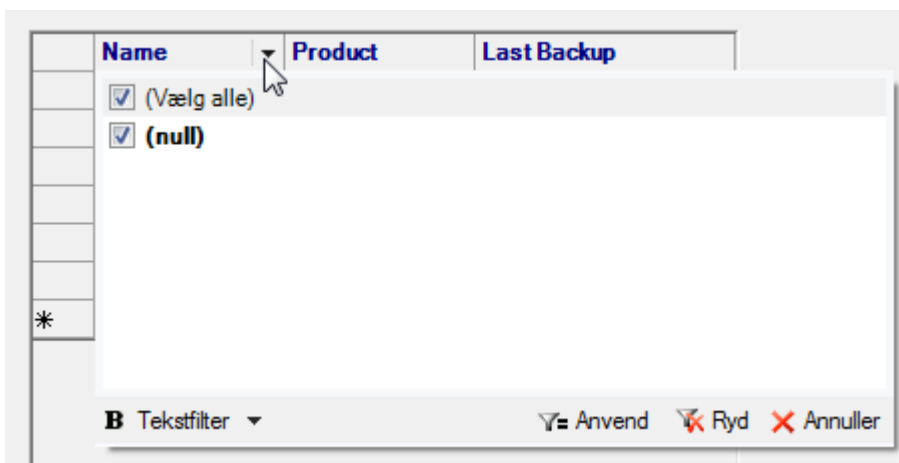
### To write code in C#

C#

```
this.c1FlexGrid1.AllowFiltering = true;  
this.c1FlexGrid1.Language = C1.Util.Localization.Language.Danish;
```

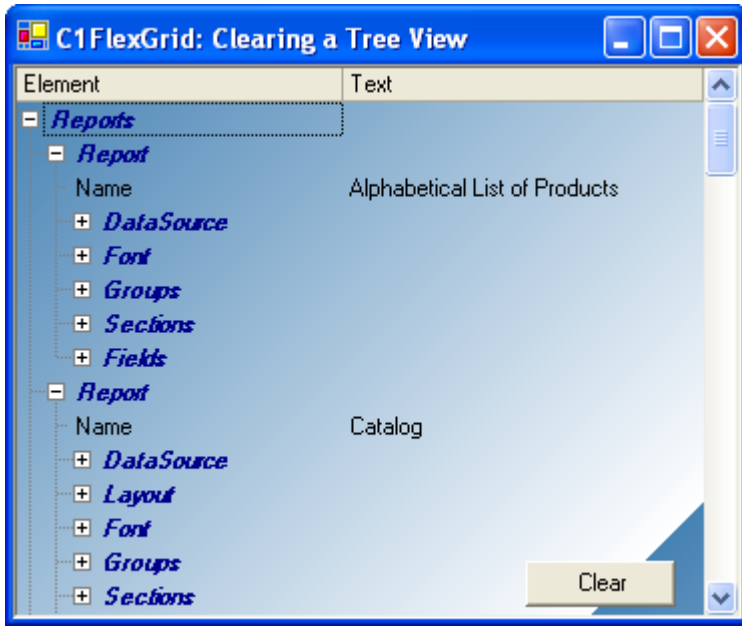
## This topic illustrates the following:

Notice the language of the column filter editor matches the language specified in the Language property.



## Clearing a Tree View

To clear a tree view in [C1FlexGrid](#), set the editable row count to zero. For additional information on creating trees, see [FlexGrid for WinForms Tutorials](#) and [Outlining and Summarizing Data](#).



Add the following code for the **Button1\_Click** event. This code will set the editable row count equal to zero, and the tree will be cleared when the **Clear** button is clicked.

#### To write code in Visual Basic

Visual Basic

```
Me.C1FlexGrid1.Rows.Count = Me.C1FlexGrid1.Rows.Fixed
```

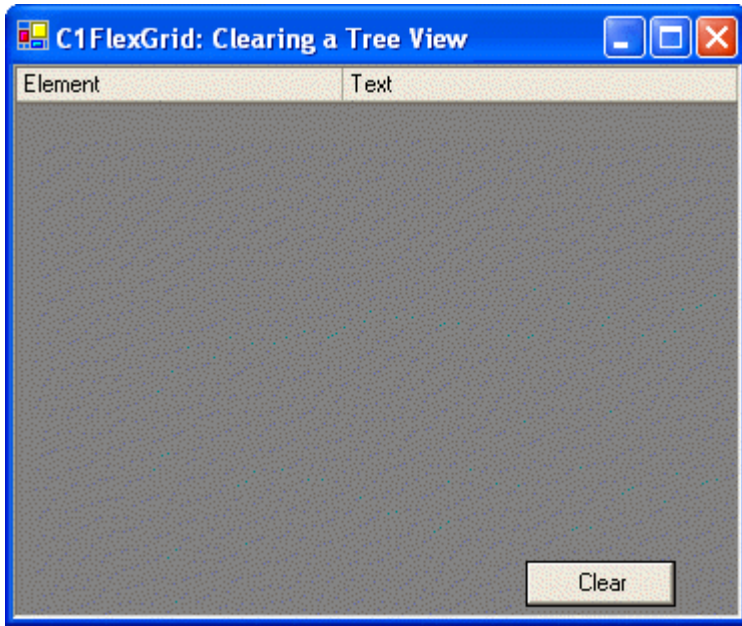
#### To write code in C#

C#

```
this.c1FlexGrid1.Rows.Count = this.c1FlexGrid1.Rows.Fixed;
```

### This topic illustrates the following:

Once the button is clicked, the tree view disappears.



## Clearing C1FlexGrid

To clear `C1FlexGrid`, use the `Clear` method. The `Clear` method clears either the content, styles, `UserData`, or all three. The following image shows the grid before any content, styles or `UserData` has been cleared.



## Clearing Content

To clear the contents of `C1FlexGrid`, add the following `Clear` method. In this example, the code was added to the `Click` event of the **Clear Contents** button.

### To write code in Visual Basic

Visual Basic

```
Me.C1FlexGrid1.Clear(C1.Win.C1FlexGrid.ClearFlags.Content)
```

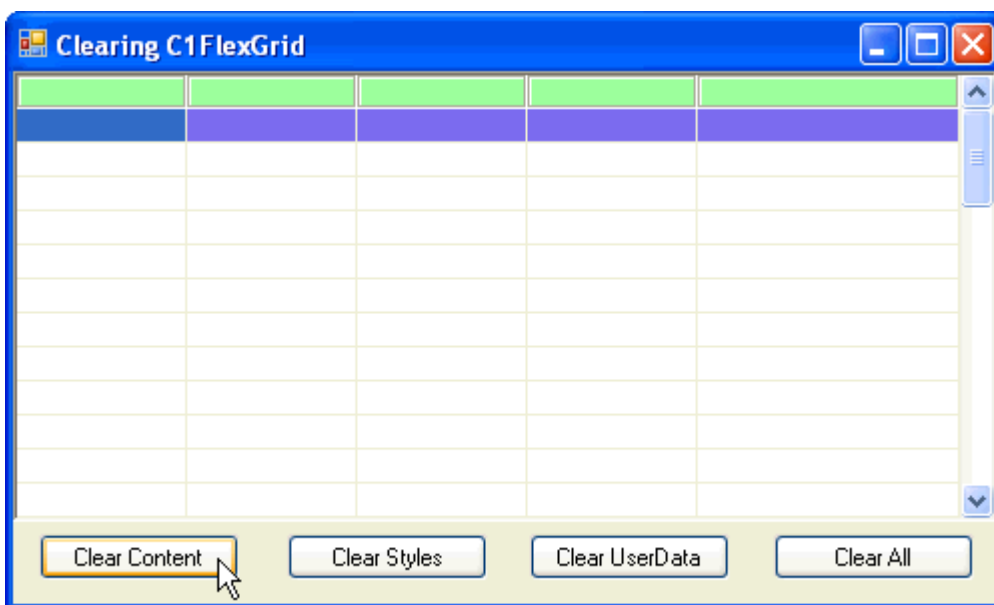
**To write code in C#**

C#

```
this.c1FlexGrid1.Clear(C1.Win.C1FlexGrid.ClearFlags.Content);
```

## This topic illustrates the following:

Clicking **Clear Content** clears only the contents, not the styles or UserData of the grid.



## Clearing Styles

To clear the style formatting of [C1FlexGrid](#), add the following [Clear](#) method. In this example, the code was added to the **Click** event of the **Clear Styles** button.

**To write code in Visual Basic**

Visual Basic

```
Me.C1FlexGrid1.Clear(C1.Win.C1FlexGrid.ClearFlags.Style)
```

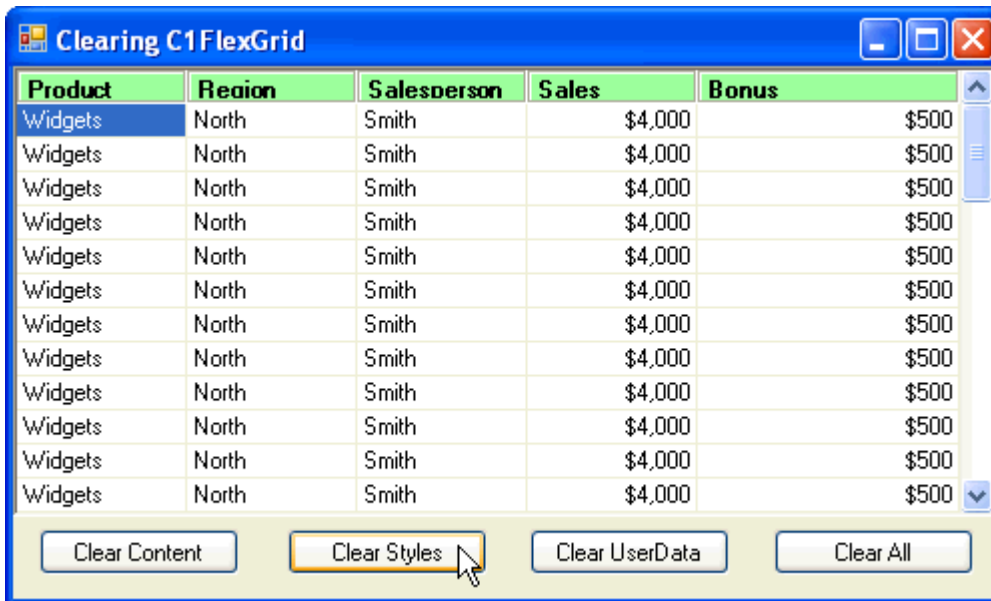
**To write code in C#**

C#

```
this.c1FlexGrid1.Clear(C1.Win.C1FlexGrid.ClearFlags.Style);
```

## This topic illustrates the following:

Clicking **Clear Styles** clears only the style formatting, not the contents or UserData of the grid.



## Clearing UserData

To clear the UserData of `C1FlexGrid`, add the following `Clear` method. In this example, the code was added to the **Click** event of the **Clear UserData** button.

### To write code in Visual Basic

Visual Basic

```
Me.C1FlexGrid1.Clear(C1.Win.C1FlexGrid.ClearFlags.UserData)
```

### To write code in C#

C#

```
this.c1FlexGrid1.Clear(C1.Win.C1FlexGrid.ClearFlags.UserData);
```

Note that there will be no visible change to the grid when clearing UserData, since UserData only stores data that is useful to the application.

## Clearing Content, Styles, and UserData

To clear the contents, style formatting, and UserData of `C1FlexGrid`, add the following `Clear` method. In this example, the code was added to the **Click** event of the **Clear All** button.

### To write code in Visual Basic

Visual Basic

```
Me.C1FlexGrid1.Clear(C1.Win.C1FlexGrid.ClearFlags.All)
```

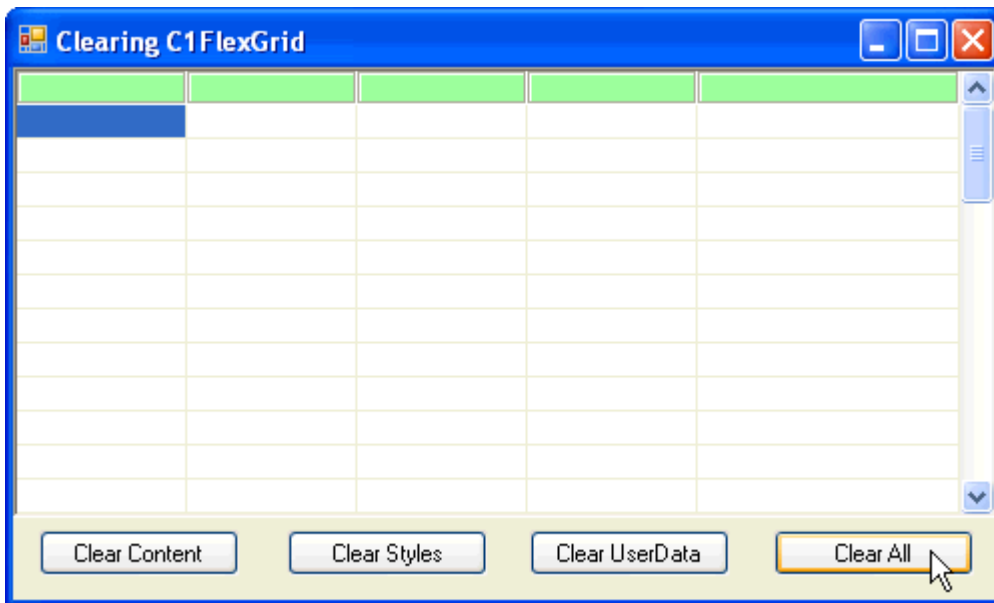
### To write code in C#

C#

```
this.c1FlexGrid1.Clear(C1.Win.C1FlexGrid.ClearFlags.All);
```

## This topic illustrates the following:

Clicking **Clear All** clears the contents, style formatting, and UserData of the grid, leaving a blank grid.



## Converting Column Letters to Uppercase

To convert column letters from lowercase to uppercase, add the following [SetupEditor](#) event to your form:

### To write code in Visual Basic

Visual Basic

```
Private Sub C1FlexGrid1_SetupEditor(ByVal sender As Object, ByVal e As  
C1.Win.C1FlexGrid.RowColEventArgs) Handles C1FlexGrid1.SetupEditor  
    If Me.C1FlexGrid1.Cols(e.Col).Name = "UCASECOL" Then  
        Dim tb As TextBox = Me.C1FlexGrid1.Editor  
        tb.CharacterCasing = CharacterCasing.Upper  
    End If  
End Sub
```

### To write code in C#

C#

```
private void c1FlexGrid1_SetupEditor(object sender, C1.Win.C1FlexGrid.RowColEventArgs  
e)  
{  
    if (this.c1FlexGrid1.Cols[e.Col].Name == "UCASECOL")  
    {
```

```

        TextBox tb = this.c1FlexGrid1.Editor as TextBox;
        tb.CharacterCasing = CharacterCasing.Upper;
    }
}

```

## Customizing Appearance Using Visual Styles

To customize the appearance of `C1FlexGrid` using Visual Styles, set the `VisualStyle` property to **Custom**, **Office2007Black**, **Office2007Blue**, **Office2007Silver**, **Office2010Blue**, **Office2010Black**, **Office2010Silver**, or **System**. This property can be set either in the designer or in code. The following table describes each of the Visual Styles:

Visual Style	Description
Custom	Do not use any visual styles. Render the control using the styles and properties only.
Office2007Black	Render the control with an appearance based on the Office 2007 Black color scheme.
Office2007Blue	Render the control with an appearance based on the Office 2007 Blue color scheme.
Office2007Silver	Render the control with an appearance based on the Office 2007 Silver color scheme.
System	Render the control with an appearance based on the current system settings.
Office2010Black	Render the control with an appearance based on the Office 2010 Black color scheme.
Office2010Silver	Render the control with an appearance based on the Office 2010 Silver color scheme.
Office2010Blue	Render the control with an appearance based on the Office 2010 Blue color scheme.

### In the Designer

Locate the `VisualStyle` property in the Properties window and set it to **Custom**, **Office2007Black**, **Office2007Blue**, **Office2007Silver**, **Office2010Blue**, **Office2010Black**, **Office2010Silver**, or **System**. In this example, the `VisualStyle` property is set to **Office2007Blue**.

### In Code

Add code to the `Form_Load` event to set the `VisualStyle` property to **Custom**, **Office2007Black**, **Office2007Blue**, **Office2007Silver**, **Office2010Blue**, **Office2010Black**, **Office2010Silver**, or **System**. The following code sets the `VisualStyle` property to **Office2007Blue**:

#### To write code in Visual Basic

Visual Basic

```
Me.C1FlexGrid1.VisualStyle = C1.Win.C1FlexGrid.VisualStyle.Office2007Blue
```



## To write code in C#

C#

```
this.c1FlexGrid1.VisualStyle = C1.Win.C1FlexGrid.VisualStyle.Office2007Blue;
```

## Custom Visual Style

No visual style is applied:

EmployeeID	LastName	FirstName
1	Davolio	Nancy
2	Fuller	Andrew
3	Leverling	Janet
4	Peacock	Margaret
5	Buchanan	Steven

## Office2007Black Visual Style

The Office 2007 Black color scheme:

EmployeeID	LastName	FirstName
1	Davolio	Nancy
2	Fuller	Andrew
3	Leverling	Janet
4	Peacock	Margaret
5	Buchanan	Steven

## Office2007Blue Visual Style

The Office 2007 Blue color scheme:

EmployeeID	LastName	FirstName
1	Davolio	Nancy
2	Fuller	Andrew
3	Leverling	Janet
4	Peacock	Margaret
5	Buchanan	Steven

## Office2007Silver Visual Style

The Office 2007 Silver color scheme.

EmployeeID	LastName	FirstName
1	Davolio	Nancy
2	Fuller	Andrew
3	Leverling	Janet
4	Peacock	Margaret
5	Buchanan	Steven

## Office2010Blue Visual Style

The Office 2010 Blue color scheme:

EmployeeID	LastName	FirstName
1	Davolio	Nancy
2	Fuller	Andrew
3	Leverling	Janet
4	Peacock	Margaret
5	Buchanan	Steven

## Office2010Silver Visual Style

The Office 2010 Silver color scheme:

EmployeeID	LastName	FirstName
1	Davolio	Nancy
2	Fuller	Andrew
3	Leverling	Janet
4	Peacock	Margaret
5	Buchanan	Steven

## Office2010Black Visual Style

The Office 2010 Black color scheme:

EmployeeID	LastName	FirstName
1	Davolio	Nancy
2	Fuller	Andrew
3	Leverling	Janet
4	Peacock	Margaret
5	Buchanan	Steven

## System Visual Style

The current system settings:

EmployeeID	LastName	FirstName
1	Davolio	Nancy
2	Fuller	Andrew
3	Leverling	Janet
4	Peacock	Margaret
5	Buchanan	Steven

## Entering Only Numbers in a Cell

To allow entering only numbers in a cell, use the [KeyPressEdit](#) event and set the *e.Handled* parameter to **True** for the invalid keys. Use the following code to set the fourth column to accept only numbers, BACKSPACE, DELETE, and the PERIOD keys:

### To write code in Visual Basic

#### Visual Basic

```
Private Sub C1FlexGrid1_KeyPressEdit(ByVal sender As Object, ByVal e As
C1.Win.C1FlexGrid.KeyPressEventArgs) Handles C1FlexGrid1.KeyPressEdit
    If e.Col = 3 Then

        ' If not the characters 0-9, PERIOD, DELETE, or BACKSPACE.
        If Not (e.KeyChar = Chr(48) Or e.KeyChar = Chr(49) Or _
            e.KeyChar = Chr(50) Or e.KeyChar = Chr(51) Or _
            e.KeyChar = Chr(52) Or e.KeyChar = Chr(53) Or _
            e.KeyChar = Chr(54) Or e.KeyChar = Chr(55) Or _
            e.KeyChar = Chr(56) Or e.KeyChar = Chr(57) Or _
            e.KeyChar = Chr(46) Or e.KeyChar = Chr(127) Or _
            e.KeyChar = Chr(8)) Then

            ' Stop the character from being entered into the
            ' control since it is an invalid key.
            e.Handled = True
        End If
    End If
End Sub
```

### To write code in C#

#### C#

```
private void c1FlexGrid1_KeyPressEdit(object sender,
C1.Win.C1FlexGrid.KeyPressEventArgs e)
{
    if( e.Col == 3 )
    {
        // If not the characters 0-9, PERIOD, DELETE, or BACKSPACE.
        if( !(e.KeyChar == 48) || (e.KeyChar == 49) ||
            (e.KeyChar == 50) || (e.KeyChar == 51) || (e.KeyChar == 52) ||
            (e.KeyChar == 53) || (e.KeyChar == 54) || (e.KeyChar == 55) ||
            (e.KeyChar == 56) || (e.KeyChar == 57) || (e.KeyChar == 46) ||
            (e.KeyChar == 127) || (e.KeyChar == 8))
```

```
        // Stop the character from being entered into the
        // control since it is an invalid key.
        e.Handled = true;
    }
}
```

## Formatting Cells

Formatting cells allows you to control the accessibility and appearance of data in a single cell or multiple cells. The following topics show you how to set cells to be read-only, format a cell to display currency values, and format cells based on their contents.

### Formatting a Cell as Read-Only

To make a single cell or multiple cells read-only, use the [BeforeEdit](#) event.

#### Single Cell Read-Only

You can make any cell in the grid read-only so that the data within it cannot be changed. For example, enter the following code to make the cell in column 1, row 1 read-only:

##### To write code in Visual Basic

###### Visual Basic

```
Private Sub C1FlexGrid1_BeforeEdit(ByVal sender As Object, ByVal e As
C1.Win.C1FlexGrid.RowColEventArgs) Handles C1FlexGrid1.BeforeEdit
    If e.Row = 1 And e.Col = 1 Then
        e.Cancel = True
    End If
End Sub
```

##### To write code in C#

###### C#

```
private void c1FlexGrid1_BeforeEdit(object sender, C1.Win.C1FlexGrid.RowColEventArgs
e)
{
    if (e.Row == 1 & e.Col == 1)
    {
        e.Cancel = true;
    }
}
```

#### Multiple Cell Read-Only

You may also want to set multiple cells in the grid to read-only at one time. Suppose you have an editable grid used to enter customers' information, such as customer IDs, addresses, and orders. While the orders and addresses may change, the customer IDs will not. The code below assumes you have nine rows in your grid, and you would like to

make all of the rows in the first column, the Customer ID column, read-only. Enter the following code:

### To write code in Visual Basic

#### Visual Basic

```
Private Sub C1FlexGrid1_BeforeEdit(ByVal sender As Object, ByVal e As C1.Win.C1FlexGrid.RowColEventArgs) Handles C1FlexGrid1.BeforeEdit
    Dim i As Integer
    For i = 1 To 9
        If e.Col = 1 And e.Row = i Then
            e.Cancel = True
        End If
    Next
End Sub
```

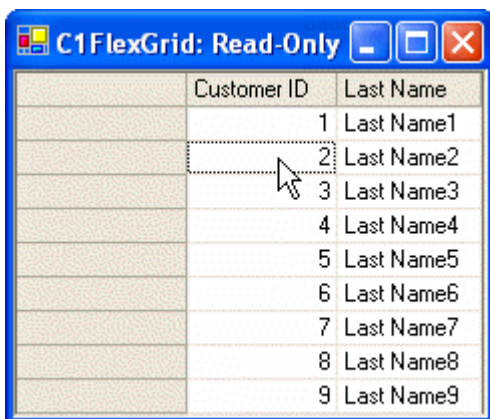
### To write code in C#

#### C#

```
private void c1FlexGrid1_BeforeEdit(object sender, C1.Win.C1FlexGrid.RowColEventArgs e)
{
    for (int i = 1; i <= 9; i++)
    {
        if (e.Col == 1 & e.Row == i)
        {
            e.Cancel = true;
        }
    }
}
```

## This topic illustrates the following:

Notice that none of the cells in the Customer ID column are editable.



## Formatting a Cell with Decimal Content

To format a cell that contains decimal numbers, set the [Format](#) property either in the designer or in code. In this example, numbers already entered in the first column will be formatted to a decimal amount.

## In the Designer

1. Select Column 1 in the grid. This will open the **Column Tasks** menu for Column 1.
2. Click the **ellipsis** button in the **Format String** box to open the **Format String** dialog box.
3. Under **Format type** select **Custom** and set the **Custom format** to **\$#,##0.00**.
4. Click **OK** to close the **Format String** dialog box.

Alternatively, the **Format** property can also be set using the **C1FlexGrid Column Editor**:

1. Open the **C1FlexGrid Column Editor**. For details on how to access the **C1FlexGrid Column Editor**, see [Accessing the C1FlexGrid Column Editor](#).
2. Select Column 1 from the right pane and set the **Format** property to **\$#,##0.00** in the left pane.
3. Click **OK** to close the editor.

## In Code


Add the following code to the **Form\_Load** event:

### To write code in Visual Basic

```
Visual Basic  
Me.C1FlexGrid1.Cols(1).Format = "$#,##0.00"
```

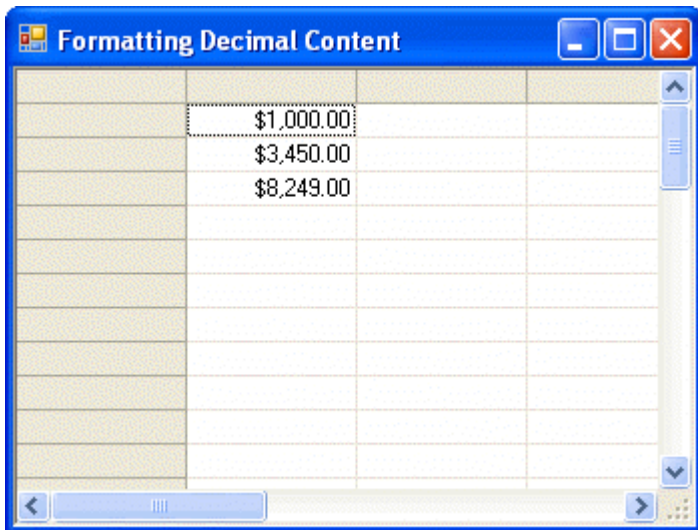
### To write code in C#

```
C#  
this.c1FlexGrid1.Cols[1].Format = "$#,##0.00";
```

 **Note:** The format **specifier** follows the standard .NET conventions. Use ',' for thousand separators and '.' for the decimal, regardless of locale.

## This topic illustrates the following:

In this example, the numbers in the first column are converted to a dollar amount.



## Formatting Cells Based on the Contents

To conditionally format cells based on the contents, create a new style and use the [CellChanged](#) event.

1. Create a new [CellStyle](#) called **LargeValue** and set the **BackColor** property to **Gold**.

### In the Designer

- Open the **C1FlexGrid Style Editor**. For details on how to access the **C1FlexGrid Style Editor**, see [Accessing the C1FlexGrid Style Editor](#).
- Click **Add** to create a new style.
- Double-click **CustomStyle1**, rename it **LargeValue**, and press ENTER when finished.
- Locate the **BackColor** property in the right pane and set it to **Gold**.
- Locate the **Font** property and click the ellipsis button next to it to open the **Font** dialog box.
- Set the **Font style** box to **Italic**.
- Click **OK** to close the **Font** dialog box.
- Click **OK** to close the **C1FlexGrid Style Editor**.

### In Code

Add the following code to the **Form\_Load** event:

#### To write code in Visual Basic

Visual Basic

```
' Create a custom style for large values.  
Dim cs As C1.Win.C1FlexGrid.CellStyle  
cs = Me.C1FlexGrid1.Styles.Add("LargeValue")  
cs.Font = New Font(Font, FontStyle.Italic)  
cs.BackColor = Color.Gold
```

#### To write code in C#

C#

```
// Create a custom style for large values.  
C1.Win.C1FlexGrid.CellStyle cs;
```

```
cs = this.c1FlexGrid1.Styles.Add("LargeValue");  
cs.Font = new Font(Font, FontStyle.Italic);  
cs.BackColor = Color.Gold;
```

2. Use the CellChanged event to format the cells based on the contents:

#### To write code in Visual Basic

##### Visual Basic

```
' Format cells based on their content.  
Private Sub C1FlexGrid1_CellChanged(ByVal sender As Object, ByVal e As  
C1.Win.C1FlexGrid.RowColEventArgs) Handles C1FlexGrid1.CellChanged  
  
    ' Mark currency values > 50,000 as LargeValues.  
    ' Reset others by setting their Style to Nothing.  
    Dim cs As C1.Win.C1FlexGridCellStyle  
    If Val(Me.C1FlexGrid1(e.Row, e.Col)) >= 50000 Then  
        cs = Me.C1FlexGrid1.Styles("LargeValue")  
    End If  
    Me.C1FlexGrid1.SetCellStyle(e.Row, e.Col, cs)  
End Sub
```

#### To write code in C#

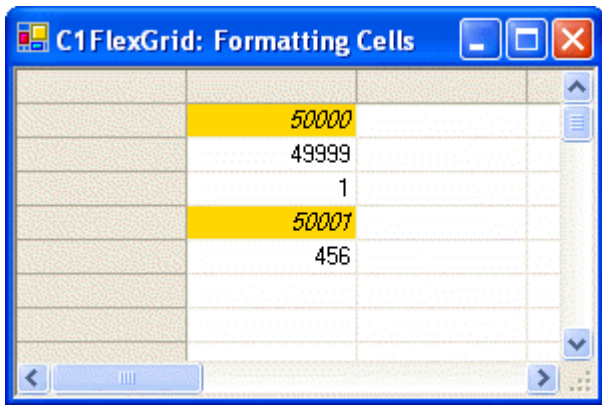
##### C#

```
// Format cells based on their content.  
private void c1FlexGrid1_CellChanged(object sender,  
C1.Win.C1FlexGrid.RowColEventArgs e)  
{  
    // Mark currency values > 50,000 as LargeValues.  
    // Reset others by setting their Style to Nothing.  
    C1.Win.C1FlexGrid.CellStyle cs;  
    if (Val(this.c1FlexGrid1[e.Row, e.Col].ToString()) >= 50000)  
    {  
        cs = this.c1FlexGrid1.Styles["LargeValue"];  
    }  
    this.c1FlexGrid1.SetCellStyle(e.Row, e.Col, cs);  
}
```

## This topic illustrates the following:

In this example, cells containing values greater than or equal to 50,000 are highlighted in gold.





## Formatting the Border Style

Formatting the border style allows you to customize the appearance of the grid. The border style can be set for both the control and the grid.

## Formatting the Border Style of the Control

To format the border style of the control, set the **ScrollableControl.BorderStyle** property to **Fixed3D**, **FixedSingle**, **Light3D**, **None**, or **XpThemes**. This property can be set either in the designer or in code. The following table describes each of the border styles.

Border	Description
<b>Fixed3D</b>	A three-dimensional border. This is the default setting.
<b>FixedSingle</b>	A single line border.
<b>Light3D</b>	A light sunken border.
<b>None</b>	No border.
<b>XpThemes</b>	A border drawn using XP themes.

## In the Designer

Locate the **BorderStyle** property in the Properties window and set it to **Fixed3D**, **FixedSingle**, **Light3D**, **None**, or **XpThemes**. In this example, the **BorderStyle** property is set to **Fixed3D**.

## In Code

Add code to the **Form\_Load** event to set the **BorderStyle** property to **Fixed3D**, **FixedSingle**, **Light3D**, **None**, or **XpThemes**. The following code sets the **BorderStyle** property to **Fixed3D**:

### To write code in Visual Basic

Visual Basic

```
Me.C1FlexGrid1.BorderStyle =
C1.Win.C1FlexGrid.Util.BaseControls.BorderStyleEnum.Fixed3D
```

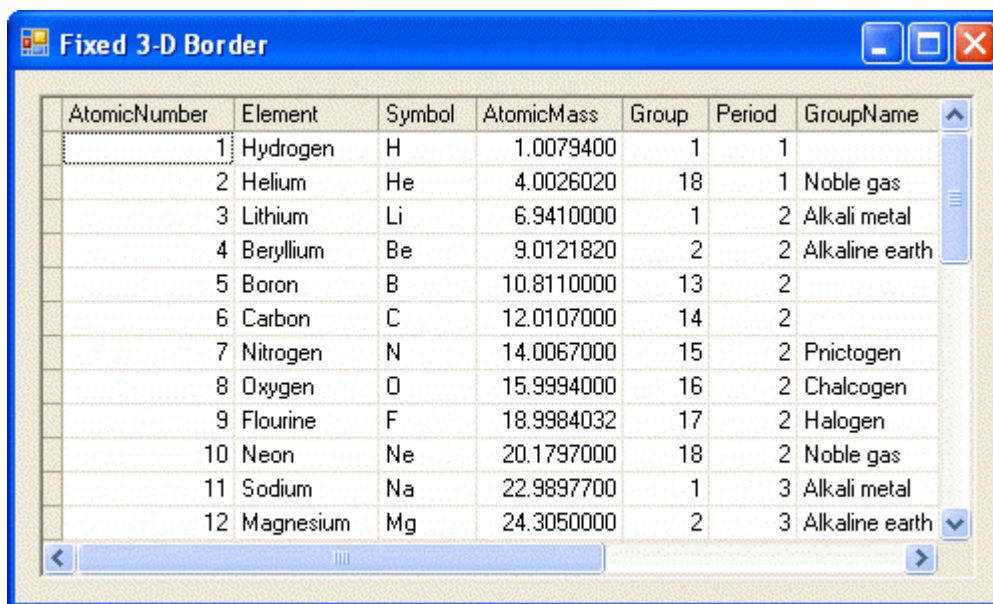
**To write code in C#**

C#

```
this.clFlexGrid1.BorderStyle =  
Cl.Win.ClFlexGrid.Util.BaseControls.BorderStyleEnum.Fixed3D;
```

## Three-Dimensional Border

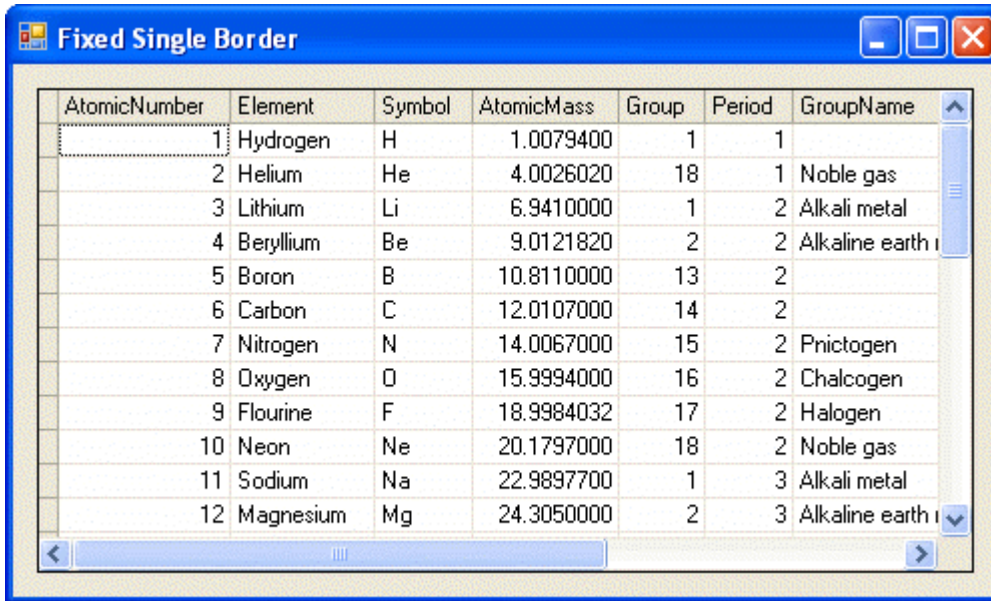
The border will be three-dimensional.



AtomicNumber	Element	Symbol	AtomicMass	Group	Period	GroupName
1	Hydrogen	H	1.0079400	1	1	
2	Helium	He	4.0026020	18	1	Noble gas
3	Lithium	Li	6.9410000	1	2	Alkali metal
4	Beryllium	Be	9.0121820	2	2	Alkaline earth
5	Boron	B	10.8110000	13	2	
6	Carbon	C	12.0107000	14	2	
7	Nitrogen	N	14.0067000	15	2	Pnictogen
8	Oxygen	O	15.9994000	16	2	Chalcogen
9	Flourine	F	18.9984032	17	2	Halogen
10	Neon	Ne	20.1797000	18	2	Noble gas
11	Sodium	Na	22.9897700	1	3	Alkali metal
12	Magnesium	Mg	24.3050000	2	3	Alkaline earth

## Single Line Border

The border will be a single line.

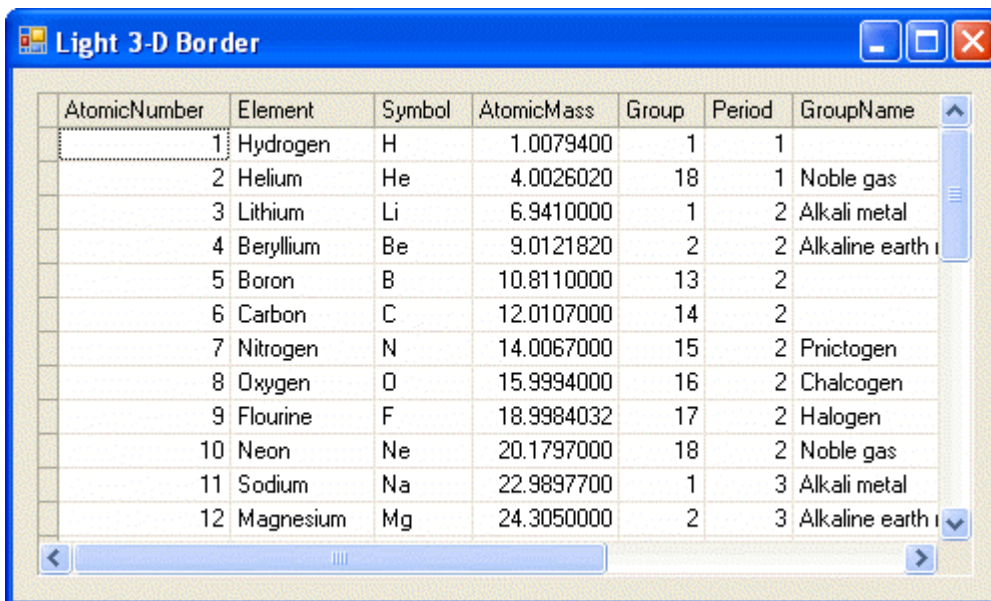


Fixed Single Border

AtomicNumber	Element	Symbol	AtomicMass	Group	Period	GroupName
1	Hydrogen	H	1.0079400	1	1	
2	Helium	He	4.0026020	18	1	Noble gas
3	Lithium	Li	6.9410000	1	2	Alkali metal
4	Beryllium	Be	9.0121820	2	2	Alkaline earth
5	Boron	B	10.8110000	13	2	
6	Carbon	C	12.0107000	14	2	
7	Nitrogen	N	14.0067000	15	2	Pnictogen
8	Oxygen	O	15.9994000	16	2	Chalcogen
9	Flourine	F	18.9984032	17	2	Halogen
10	Neon	Ne	20.1797000	18	2	Noble gas
11	Sodium	Na	22.9897700	1	3	Alkali metal
12	Magnesium	Mg	24.3050000	2	3	Alkaline earth

## Light Sunken Border

The border will be light and three-dimensional.

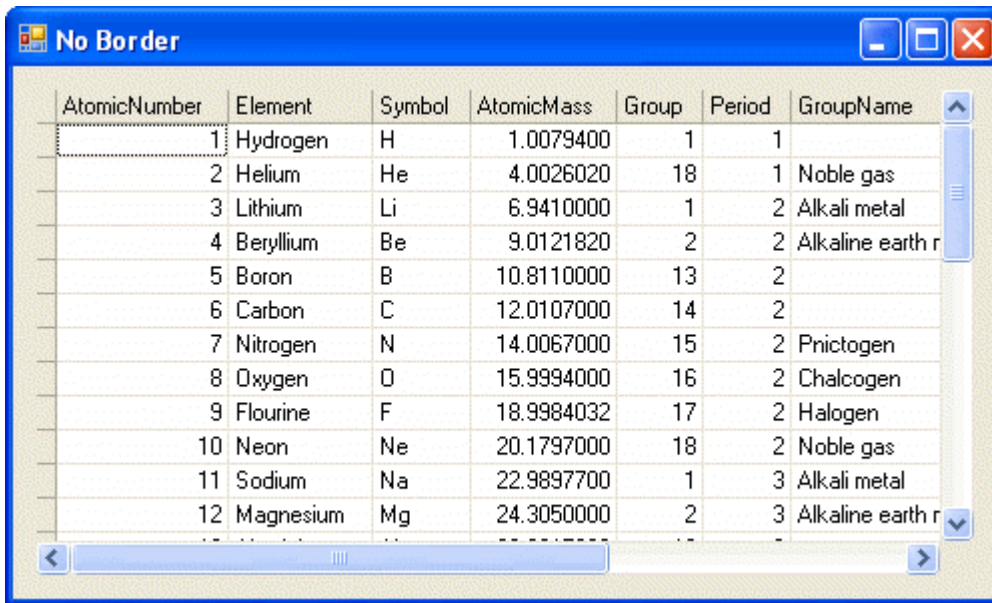


Light 3-D Border

AtomicNumber	Element	Symbol	AtomicMass	Group	Period	GroupName
1	Hydrogen	H	1.0079400	1	1	
2	Helium	He	4.0026020	18	1	Noble gas
3	Lithium	Li	6.9410000	1	2	Alkali metal
4	Beryllium	Be	9.0121820	2	2	Alkaline earth
5	Boron	B	10.8110000	13	2	
6	Carbon	C	12.0107000	14	2	
7	Nitrogen	N	14.0067000	15	2	Pnictogen
8	Oxygen	O	15.9994000	16	2	Chalcogen
9	Flourine	F	18.9984032	17	2	Halogen
10	Neon	Ne	20.1797000	18	2	Noble gas
11	Sodium	Na	22.9897700	1	3	Alkali metal
12	Magnesium	Mg	24.3050000	2	3	Alkaline earth

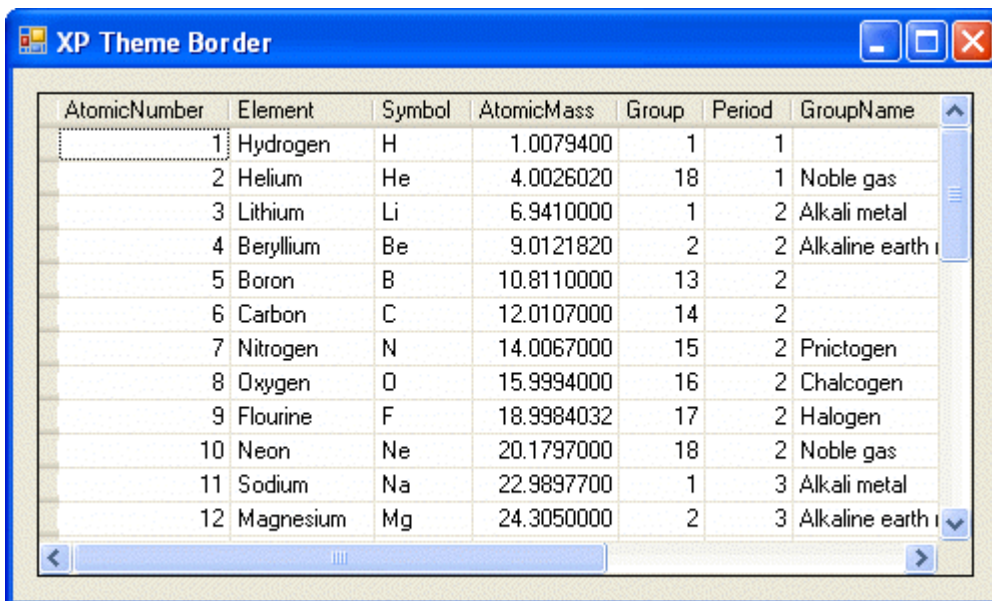
## No Border

There is no border.



## XP Theme Border

The border will be XP themed.



## Formatting the Border Style of the Grid

To format the border style of the grid, set the [Style](#) property to **Dotted**, **Double**, **Fillet**, **Flat**, **Groove**, **Inset**, **None**, or **Raised**. This property can be set using [C1FlexGrids](#) built-in styles either in the designer or in code. The following table describes each of the border styles.

Border	Description
<b>Dotted</b>	Dotted border.

Border	Description
<b>Double</b>	Double border.
<b>Fillet</b>	Fillet border.
<b>Flat</b>	Solid flat border.
<b>Groove</b>	Groove border.
<b>Inset</b>	Inset border.
<b>None</b>	No border. This is the default setting.
<b>Raised</b>	Raised border.

## In the Designer

1. Open the **C1FlexGrid Style Editor**. For details on how to access the **C1FlexGrid Style Editor**, see [Accessing the C1FlexGrid Style Editor](#).
2. Select **Normal** under **Built-In Styles**. In the right column, locate the **Border** property and expand it. Set the **Style** property to **Dotted**, **Double**, **Fillet**, **Flat**, **Groove**, **Inset**, **None**, or **Raised**. In this example, the **Style** property is set to **Dotted**.
3. Click **OK** to close the editor.

## In Code

### To write code in Visual Basic

Visual Basic

```
Me.C1FlexGrid1.Styles("Normal").Border.Style =  
C1.Win.C1FlexGrid.BorderStyleEnum.Dotted
```

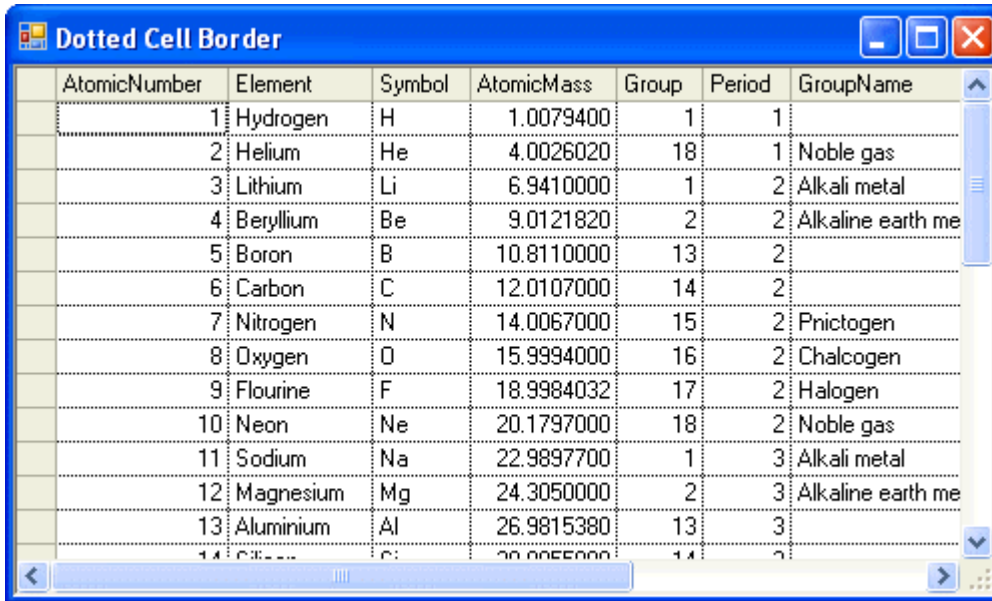
### To write code in C#

C#

```
this.c1FlexGrid1.Styles["Normal"].Border.Style =  
C1.Win.C1FlexGrid.BorderStyleEnum.Dotted;
```

## Dotted Border

The cell border will be a dotted line.

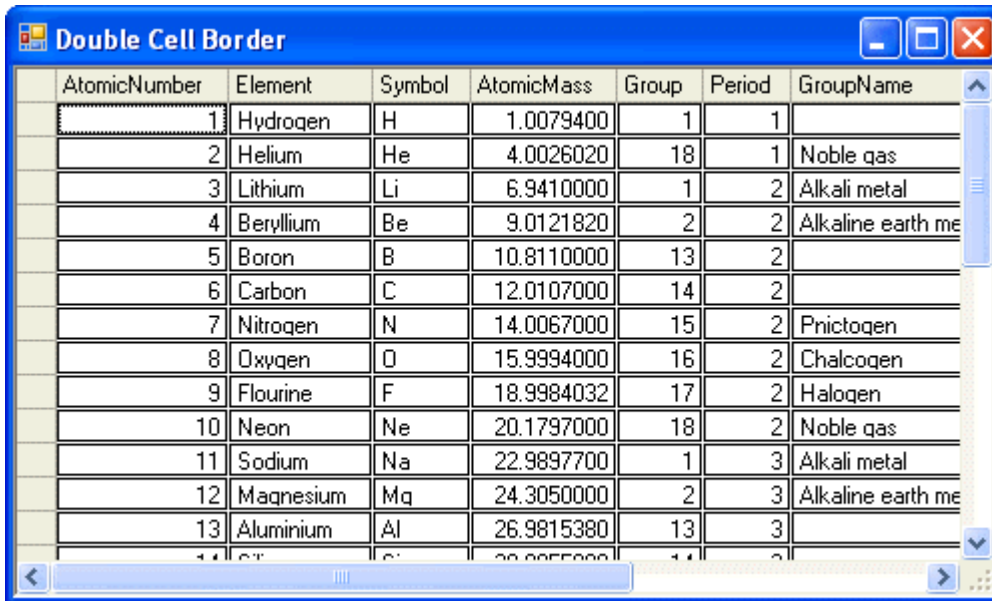


The screenshot shows a window titled "Dotted Cell Border" containing a table with a dotted border. The table has the following data:

AtomicNumber	Element	Symbol	AtomicMass	Group	Period	GroupName
1	Hydrogen	H	1.0079400	1	1	
2	Helium	He	4.0026020	18	1	Noble gas
3	Lithium	Li	6.9410000	1	2	Alkali metal
4	Beryllium	Be	9.0121820	2	2	Alkaline earth me
5	Boron	B	10.8110000	13	2	
6	Carbon	C	12.0107000	14	2	
7	Nitrogen	N	14.0067000	15	2	Pnictogen
8	Oxygen	O	15.9994000	16	2	Chalcoqen
9	Flourine	F	18.9984032	17	2	Haloqen
10	Neon	Ne	20.1797000	18	2	Noble gas
11	Sodium	Na	22.9897700	1	3	Alkali metal
12	Magnesium	Mg	24.3050000	2	3	Alkaline earth me
13	Aluminium	Al	26.9815380	13	3	

## Double Border

The cell border will be double lines.

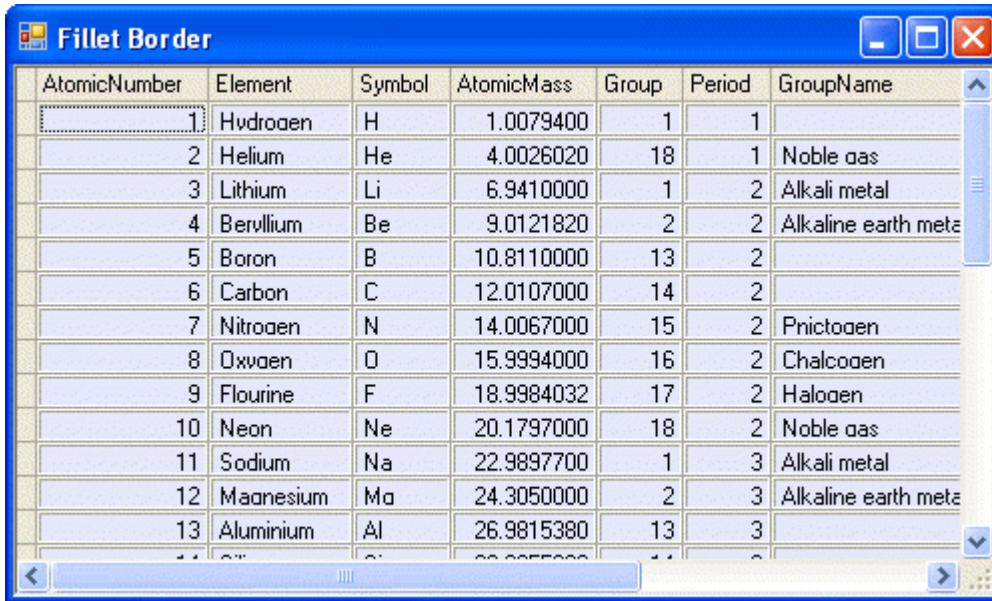


The screenshot shows a window titled "Double Cell Border" containing a table with a double border. The table has the following data:

AtomicNumber	Element	Symbol	AtomicMass	Group	Period	GroupName
1	Hydrogen	H	1.0079400	1	1	
2	Helium	He	4.0026020	18	1	Noble gas
3	Lithium	Li	6.9410000	1	2	Alkali metal
4	Beryllium	Be	9.0121820	2	2	Alkaline earth me
5	Boron	B	10.8110000	13	2	
6	Carbon	C	12.0107000	14	2	
7	Nitrogen	N	14.0067000	15	2	Pnictogen
8	Oxygen	O	15.9994000	16	2	Chalcoqen
9	Flourine	F	18.9984032	17	2	Haloqen
10	Neon	Ne	20.1797000	18	2	Noble gas
11	Sodium	Na	22.9897700	1	3	Alkali metal
12	Magnesium	Mg	24.3050000	2	3	Alkaline earth me
13	Aluminium	Al	26.9815380	13	3	

## Fillet Border

The cell border will be fillet.

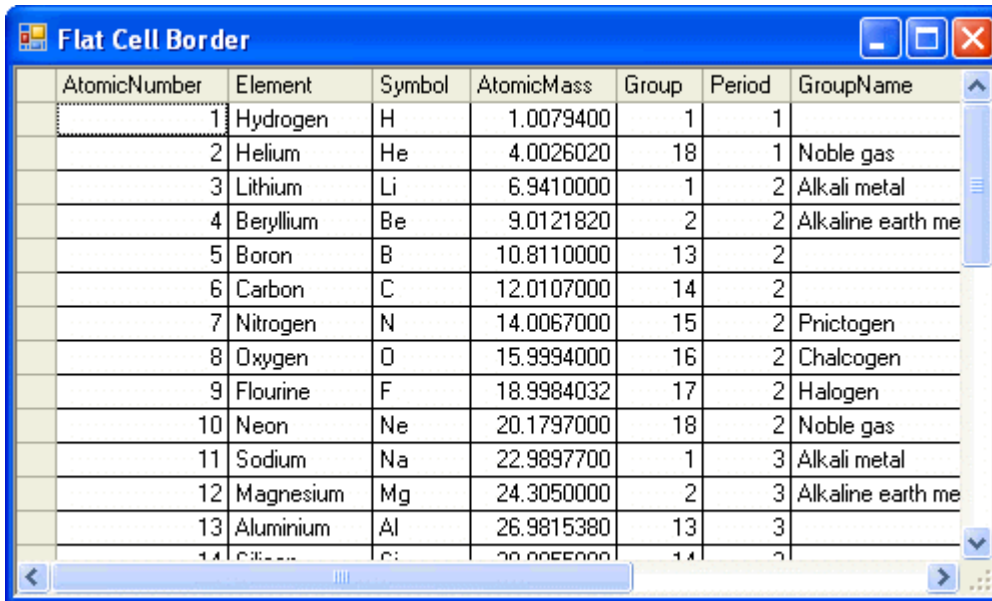


The screenshot shows a window titled "Fillet Border" containing a grid with 7 columns: AtomicNumber, Element, Symbol, AtomicMass, Group, Period, and GroupName. The grid displays the first 13 elements of the periodic table. The cell borders are rounded (fillet style).

AtomicNumber	Element	Symbol	AtomicMass	Group	Period	GroupName
1	Hydrogen	H	1.0079400	1	1	
2	Helium	He	4.0026020	18	1	Noble gas
3	Lithium	Li	6.9410000	1	2	Alkali metal
4	Beryllium	Be	9.0121820	2	2	Alkaline earth meta
5	Boron	B	10.8110000	13	2	
6	Carbon	C	12.0107000	14	2	
7	Nitrogen	N	14.0067000	15	2	Pnictogen
8	Oxygen	O	15.9994000	16	2	Chalcogen
9	Flourine	F	18.9984032	17	2	Halogen
10	Neon	Ne	20.1797000	18	2	Noble gas
11	Sodium	Na	22.9897700	1	3	Alkali metal
12	Maanesium	Ma	24.3050000	2	3	Alkaline earth meta
13	Aluminium	Al	26.9815380	13	3	

## Flat Border

The cell border will be a solid flat line.

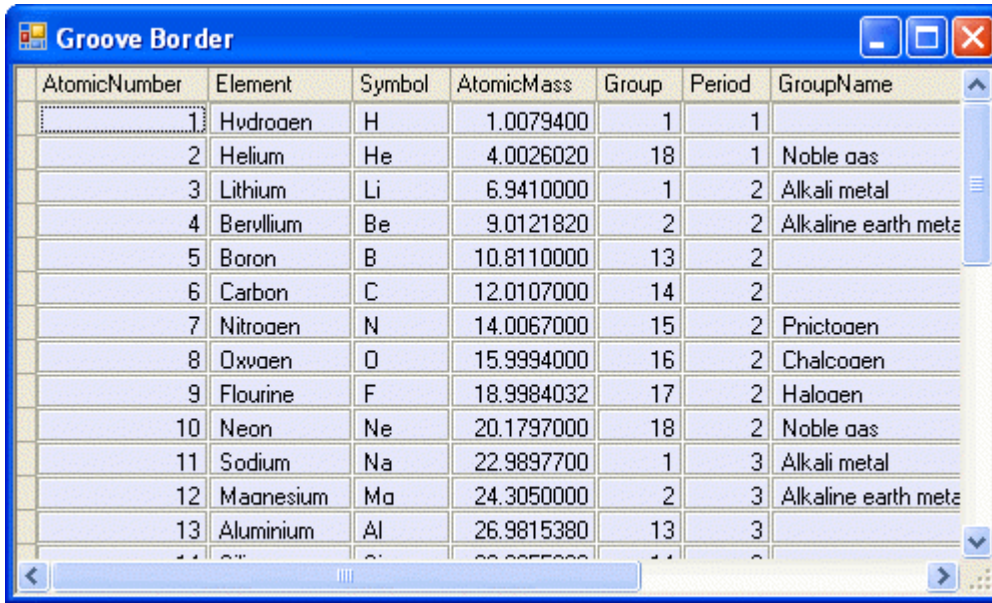


The screenshot shows a window titled "Flat Cell Border" containing a grid with the same 7 columns as the previous image. The grid displays the first 13 elements of the periodic table. The cell borders are solid and flat.

AtomicNumber	Element	Symbol	AtomicMass	Group	Period	GroupName
1	Hydrogen	H	1.0079400	1	1	
2	Helium	He	4.0026020	18	1	Noble gas
3	Lithium	Li	6.9410000	1	2	Alkali metal
4	Beryllium	Be	9.0121820	2	2	Alkaline earth me
5	Boron	B	10.8110000	13	2	
6	Carbon	C	12.0107000	14	2	
7	Nitrogen	N	14.0067000	15	2	Pnictogen
8	Oxygen	O	15.9994000	16	2	Chalcogen
9	Flourine	F	18.9984032	17	2	Halogen
10	Neon	Ne	20.1797000	18	2	Noble gas
11	Sodium	Na	22.9897700	1	3	Alkali metal
12	Magnesium	Mg	24.3050000	2	3	Alkaline earth me
13	Aluminium	Al	26.9815380	13	3	

## Groove Border

The cell border will be grooved.

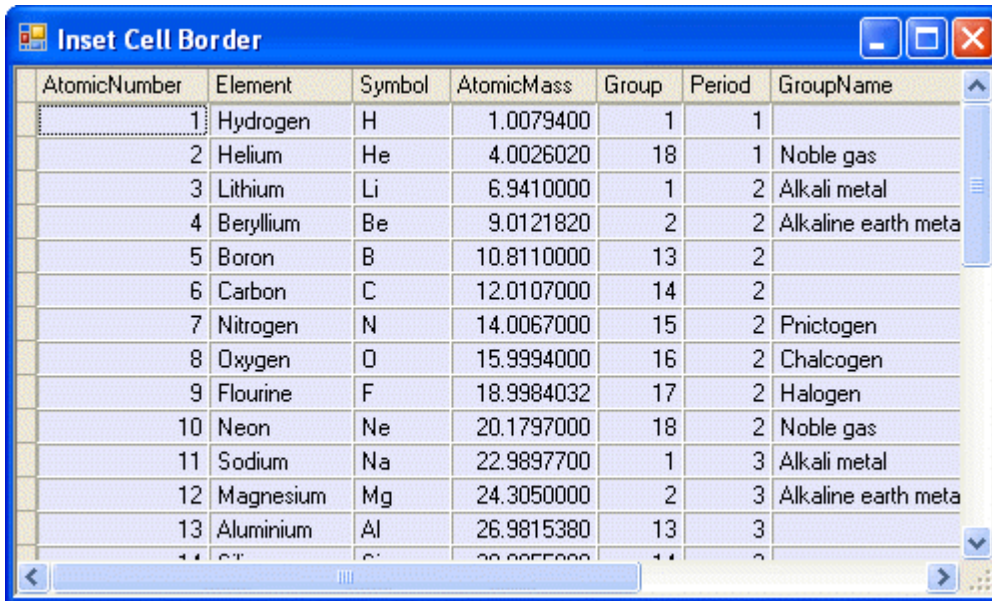


**Groove Border**

AtomicNumber	Element	Symbol	AtomicMass	Group	Period	GroupName
1	Hydrogen	H	1.0079400	1	1	
2	Helium	He	4.0026020	18	1	Noble gas
3	Lithium	Li	6.9410000	1	2	Alkali metal
4	Beryllium	Be	9.0121820	2	2	Alkaline earth meta
5	Boron	B	10.8110000	13	2	
6	Carbon	C	12.0107000	14	2	
7	Nitrogen	N	14.0067000	15	2	pnictogen
8	Oxygen	O	15.9994000	16	2	Chalcogen
9	Flourine	F	18.9984032	17	2	Halogen
10	Neon	Ne	20.1797000	18	2	Noble gas
11	Sodium	Na	22.9897700	1	3	Alkali metal
12	Maanesium	Ma	24.3050000	2	3	Alkaline earth meta
13	Aluminium	Al	26.9815380	13	3	

## Inset Border

The cell border will be inset.



**Inset Cell Border**

AtomicNumber	Element	Symbol	AtomicMass	Group	Period	GroupName
1	Hydrogen	H	1.0079400	1	1	
2	Helium	He	4.0026020	18	1	Noble gas
3	Lithium	Li	6.9410000	1	2	Alkali metal
4	Beryllium	Be	9.0121820	2	2	Alkaline earth meta
5	Boron	B	10.8110000	13	2	
6	Carbon	C	12.0107000	14	2	
7	Nitrogen	N	14.0067000	15	2	pnictogen
8	Oxygen	O	15.9994000	16	2	Chalcogen
9	Flourine	F	18.9984032	17	2	Halogen
10	Neon	Ne	20.1797000	18	2	Noble gas
11	Sodium	Na	22.9897700	1	3	Alkali metal
12	Magnesium	Mg	24.3050000	2	3	Alkaline earth meta
13	Aluminium	Al	26.9815380	13	3	

## No Border

There is no cell border.



The screenshot shows a window titled "No Border" containing a FlexGrid. The grid displays a table of elements with the following columns: AtomicNumber, Element, Symbol, AtomicMass, Group, Period, and GroupName. The first few rows are visible, showing elements from Hydrogen to Aluminium. The grid has a standard Windows-style border and scrollbars, but the individual cells within the grid do not have borders.

AtomicNumber	Element	Symbol	AtomicMass	Group	Period	GroupName
1	Hydrogen	H	1.0079400	1	1	
2	Helium	He	4.0026020	18	1	Noble gas
3	Lithium	Li	6.9410000	1	2	Alkali metal
4	Beryllium	Be	9.0121820	2	2	Alkaline earth meta
5	Boron	B	10.8110000	13	2	
6	Carbon	C	12.0107000	14	2	
7	Nitrogen	N	14.0067000	15	2	Prictogen
8	Oxygen	O	15.9994000	16	2	Chalcogen
9	Flourine	F	18.9984032	17	2	Halogen
10	Neon	Ne	20.1797000	18	2	Noble gas
11	Sodium	Na	22.9897700	1	3	Alkali metal
12	Magnesium	Mg	24.3050000	2	3	Alkaline earth meta
13	Aluminium	Al	26.9815380	13	3	

## Raised Border

The cell border will be raised.

The screenshot shows a window titled "Raised Cell Border" containing a FlexGrid. The grid displays the same table of elements as the previous screenshot. In this version, each cell in the grid has a raised border, giving it a 3D appearance. The window title, controls, and scrollbars are the same as in the "No Border" example.

AtomicNumber	Element	Symbol	AtomicMass	Group	Period	GroupName
1	Hydrogen	H	1.0079400	1	1	
2	Helium	He	4.0026020	18	1	Noble gas
3	Lithium	Li	6.9410000	1	2	Alkali metal
4	Beryllium	Be	9.0121820	2	2	Alkaline earth meta
5	Boron	B	10.8110000	13	2	
6	Carbon	C	12.0107000	14	2	
7	Nitrogen	N	14.0067000	15	2	Prictogen
8	Oxygen	O	15.9994000	16	2	Chalcogen
9	Flourine	F	18.9984032	17	2	Halogen
10	Neon	Ne	20.1797000	18	2	Noble gas
11	Sodium	Na	22.9897700	1	3	Alkali metal
12	Magnesium	Mg	24.3050000	2	3	Alkaline earth meta
13	Aluminium	Al	26.9815380	13	3	

## Freezing Rows and Columns

To allow the user to freeze rows and columns with the mouse, set the [AllowFreezing](#) property to **Columns** to freeze only columns, **Rows** to freeze only rows, or **Both** to freeze both columns and rows. Conversely, to disable freezing, set the AllowFreezing property to **None**, which is the default setting. This property can be set either in the designer or in code.

## In the Designer

Locate the **AllowFreezing** property in the Properties window and set it to **Both**.

## In Code

Add the following code to the **Form\_Load** event to set the **AllowFreezing** property to **Both**:

### To write code in Visual Basic

Visual Basic



```
Me.C1FlexGrid1.AllowFreezing = C1.Win.C1FlexGrid.AllowFreezingEnum.Both
```

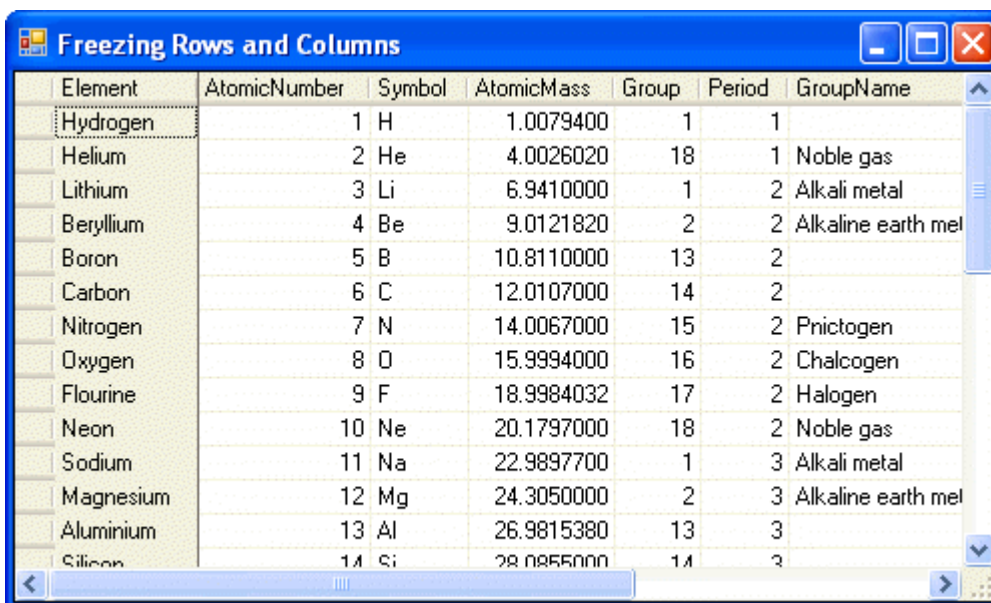
### To write code in C#

C#

```
this.c1FlexGrid1.AllowFreezing = C1.Win.C1FlexGrid.AllowFreezingEnum.Both;
```

## This topic illustrates the following:

When the mouse pointer becomes the lock rows  or the lock columns  icon, click and drag the mouse over the rows or columns to freeze. In this example, the *Element* column is frozen and will remain on the grid when it is scrolled to the right.



In this example, the row containing Hydrogen is frozen and will remain on the grid when it is scrolled to the bottom.

Element	Period	GroupName	Block	StandardState	Classification
Hydrogen	1	1	s-block	Gas	Non-Metallic
Argon	18	3 Noble gas	p-block	Gas	Non-Metallic
Potassium	1	4 Alkali metal	s-block	Solid	Metallic
Calcium	2	4 Alkaline earth metal	s-block	Solid	Metallic
Scandium	3		d-block	Solid	Metallic
Titanium	4		d-block	Solid	Metallic
Vanadium	5		d-block	Solid	Metallic
Chromium	6		d-block	Solid	Metallic
Manganese	7		d-block	Solid	Metallic
Iron	8		d-block	Solid	Metallic
Cobalt	9		d-block	Solid	Metallic
Nickel	10		d-block	Solid	Metallic
Copper	11	4 Coinage metal	d-block	Solid	Metallic
Zinc	12	4	d-block	Solid	Metallic

**Note:** Setting the **AllowFreezing** property to **Both** allows both rows and columns to be frozen at the same time, as seen in the following image.

## Getting the Width of a Partially Visible Column

If you need to obtain the width of the visible part of a column that is not fully visible due to horizontal scrolling, you can easily do so with just a few lines of code.

Suppose you have a **C1FlexGrid** with multiple columns and a **Label** control on the form. Add the following code to the **Form\_Load** event:

### To write code in Visual Basic

#### Visual Basic

```
Dim lastCol As Integer = C1FlexGrid1.RightCol
    Dim colPoint As Point = C1FlexGrid1.GetCellRect(1, lastCol).Location
    Dim flexPoint As Point = New Point(C1FlexGrid1.ClientRectangle.Right,
C1FlexGrid1.ClientRectangle.Top)

    Label1.Text = flexPoint.X - colPoint.X
```

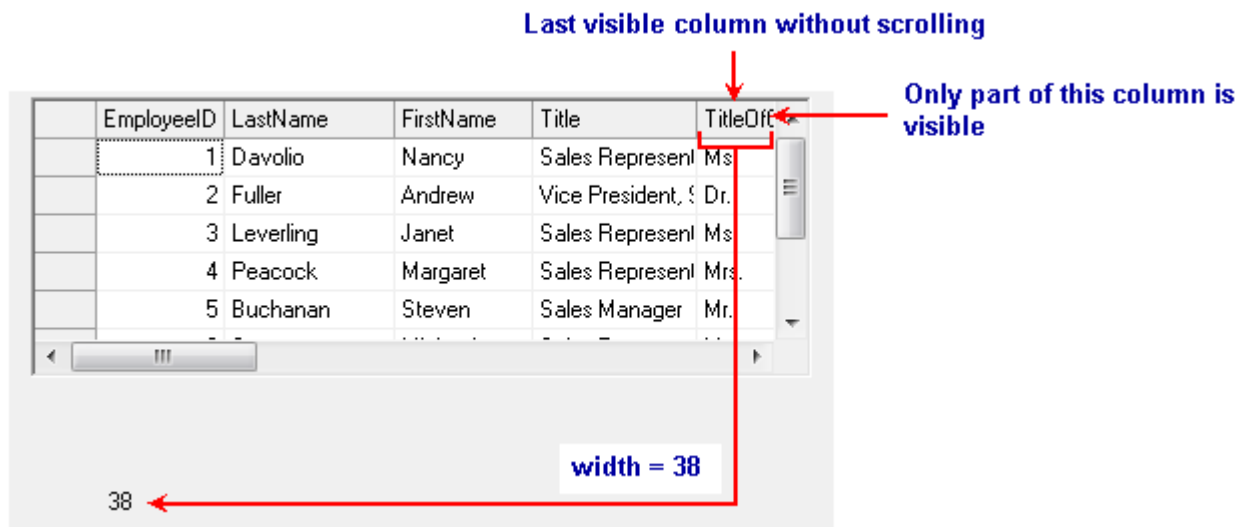
### To write code in C#

#### C#

```
int lastCol = c1FlexGrid1.RightCol;
Point colPoint = c1FlexGrid1.GetCellRect(1, lastCol).Location;
Point flexPoint = new Point(c1FlexGrid1.ClientRectangle.Right,
c1FlexGrid1.ClientRectangle.Top);

label1.Text = flexPoint.X - colPoint.X;}
```

When you run the project, the label will show the width of the visible part of the last visible column. In this case, the **TitleOfCourtesy** column is only partially visible. The width of the part of that column that is showing is **38**.



## Loading and Saving Open XML Files

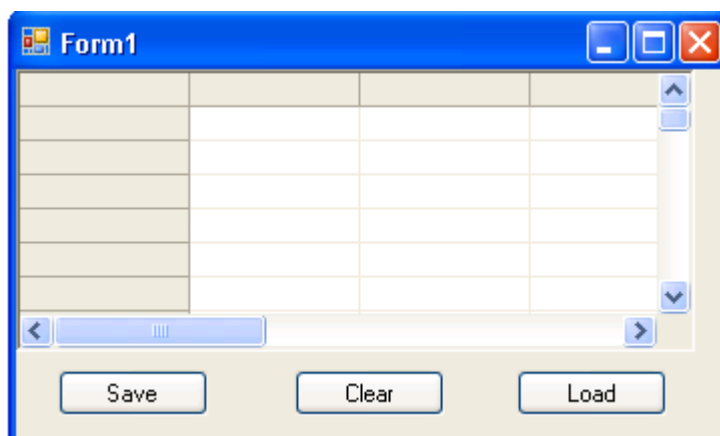
You can load and save Microsoft Excel 2007 Open XML files within a [C1FlexGrid](#) control. Open XML is an open, standards-based format introduced by Microsoft in Office 2007. Open XML files contain a number of XML files compressed using Zip compression. Because they are compressed, Open XML files are usually much smaller than traditional document files (such as XLS files).

By default, the **Load** and **Save** methods in **C1FlexGrid** select the appropriate file format automatically based on the file extension. Any files with a "XLSX" or "ZIP" extensions are treated as Open XML files.

The [LoadGrid](#) and [SaveGrid](#) methods also have overloads that take the file type as a parameter. This allows you to take control over the file format and not rely on the file extension. For example, your application may use the Open XML format for its data files, but with an extension other than "XLSX". You can specify the file type as "Excel" and use the *OpenXml* option of the [FileFlags](#) enumeration.

### To save and load a grid as an OpenXml file, complete the following steps:

1. Add three buttons to the form containing the C1FlexGrid control and enter **Save**, **Clear**, and **Load** in the **Text** property for each, respectively, in the Visual Basic Properties window.



- From the Designer, double-click the **Save** button, **Button\_1**, and enter code in the **Button1\_Click** event, so it looks like the following:

**To write code in Visual Basic**

## Visual Basic

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click
    Me.C1FlexGrid1.SaveGrid("C:\test\myfile.xlsx", FileFormatEnum.Excel,
FileFlags.OpenXml)
End Sub
```

**To write code in C#**

## C#

```
private void button1_Click(object sender, EventArgs e)
{
    this.c1FlexGrid1.SaveGrid(@"C:\test\myfile.xlsx",
FileFormatEnum.Excel, FileFlags.OpenXml);
}
```



**Note:** You must have the Imports or using statement at the top of your form in order for this code to work correctly. If coding in Visual Basic, use Imports C1.Win.C1FlexGrid. If coding in C#, use using C1.Win.C1FlexGrid; For more information, see the Namespaces topic.

- Enter the following code for the **Button2\_Click** event to be used for the **Clear** button:

**To write code in Visual Basic**

## Visual Basic

```
Private Sub Button2_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button2.Click
    Me.C1FlexGrid1.Clear(ClearFlags.All)
End Sub
```

**To write code in C#**

## C#

```
private void button2_Click(object sender, EventArgs e)
{
    this.c1FlexGrid1.Clear(ClearFlags.All);
}
```

- Enter the following code for the **Button3\_Click** event to be used for the **Load** button:

**To write code in Visual Basic**

## Visual Basic

```
Private Sub button3_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button3.Click
    Me.C1FlexGrid1.LoadGrid("C:\test\myfile.xlsx", FileFormatEnum.Excel,
```

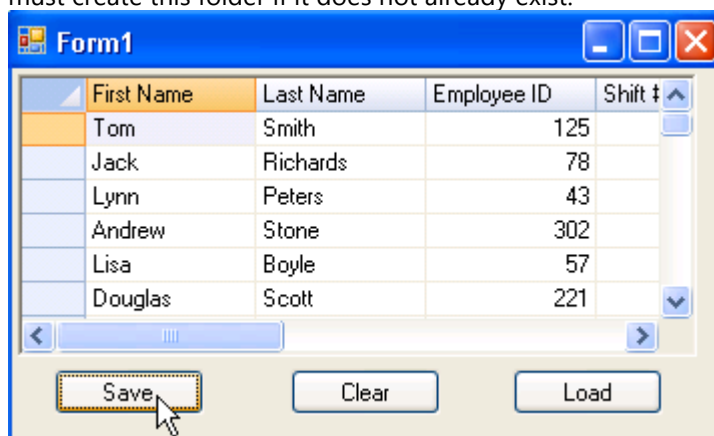
```
FileFlags.OpenXml)
End Sub
```

### To write code in C#

```
C#
private void button3_Click(object sender, EventArgs e)
{
    this.c1FlexGrid1.LoadGrid(@"C:\test\myfile.xlsx",
FileFormatEnum.Excel, FileFlags.OpenXml);
}
```

## Run the project:

1. Enter content in the grid, as desired.
2. Click the **Save** button. The grid is saved as an Open XML file with an .xlsx extension to the "C:\test" folder. You must create this folder if it does not already exist.



3. Click the **Clear** button to clear the grid.
4. Click the **Load** button, and the grid you previously saved is loaded.

## Populating an Unbound Grid with Data

To populate an unbound grid with data, use the grid's indexer to populate a column, row, or a single cell. To populate a cell range, use the [Data](#) property.

## Populating a Column with Data

To populate a column with data when the form loads, set a loop with the grid's indexer to the data to fill the column. Add the following code to the **Form\_Load** event to set column one to zero:

### To write code in Visual Basic

```
Visual Basic
Dim r As Integer
For r = C1FlexGrid1.Rows.Fixed To C1FlexGrid1.Rows.Count - 1
    Me.C1FlexGrid1(r, 1) = 0
```

Next

#### To write code in C#

```
C#  
  
int r;  
for (r = c1FlexGrid1.Rows.Fixed; r <= c1FlexGrid1.Rows.Count -1;r++)  
{  
    this.c1FlexGrid1[r, 1] = 0;  
}
```

### This topic illustrates the following:

Column one fills with zeros.

## Populating a Range of Cells with Data

To populate a range of cells with data when the form loads, set the [Data](#) property of the CellRange. Add the following code to the **Form\_Load** event to set the **Data** property to zero:

#### To write code in Visual Basic

```
Visual Basic  
  
Dim rng As C1.Win.C1FlexGrid.CellRange = Me.C1FlexGrid1.GetCellRange(1, 1, 3, 3)  
rng.Data = 0
```

#### To write code in C#

```
C#  
  
C1.Win.C1FlexGrid.CellRange rng = this.c1FlexGrid1.GetCellRange(1,1,3,3);  
rng.Data = 0;
```

### This topic illustrates the following:

The cell range fills with zeros.

## Populating a Row with Data

To populate a row with data when the form loads, set a loop with the grid's indexer to the data to fill the row. Add the following code to the **Form\_Load** event to set column one to zero:

#### To write code in Visual Basic

```
Visual Basic  
  
Dim r As Integer  
For r = C1FlexGrid1.Cols.Fixed To C1FlexGrid1.Cols.Count - 1  
    Me.C1FlexGrid1(1, r) = 0  
Next
```

**To write code in C#**

```
C#  
  
int r;  
for (r = c1FlexGrid1.Cols.Fixed; r <= c1FlexGrid1.Cols.Count -1;r++)  
{  
    this.c1FlexGrid1[1, r] = 0;  
}
```

**This topic illustrates the following:**

Row one fills with zeros.

## Populating a Single Cell with Data

To populate a single cell with data when the form loads, set the grid's indexer to the data. Add the following code to the **Form\_Load** event to set the cell text:

**To write code in Visual Basic**

```
Visual Basic  
  
Me.C1FlexGrid1(3, 2) = "Cell Text"
```

**To write code in C#**

```
C#  
  
this.c1FlexGrid1[3, 2] = "Cell Text";
```

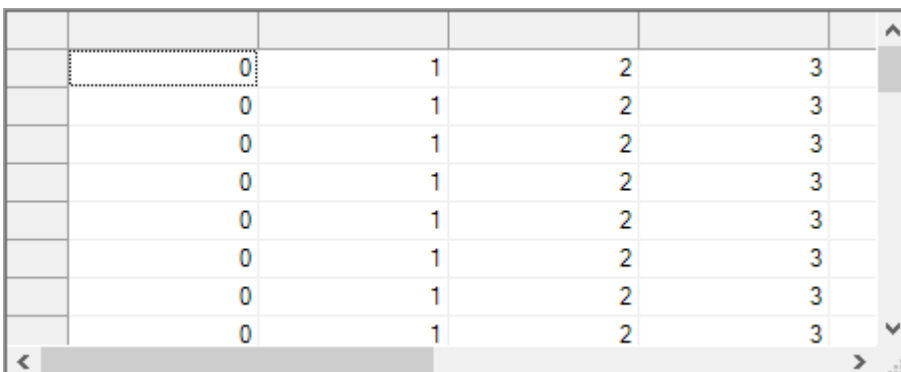
**This topic illustrates the following:**

The text **Cell Text** is in the fourth row, third column.

## Transposing Data in Grid

Transposing data refers to swapping column data and row data. In FlexGrid, this can be achieved using [Transpose\(\)](#) method.

For example, you want to transpose column and row data in the grid originally displaying data as follows:



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



Add the following line of code after the code to populate the grid with data:

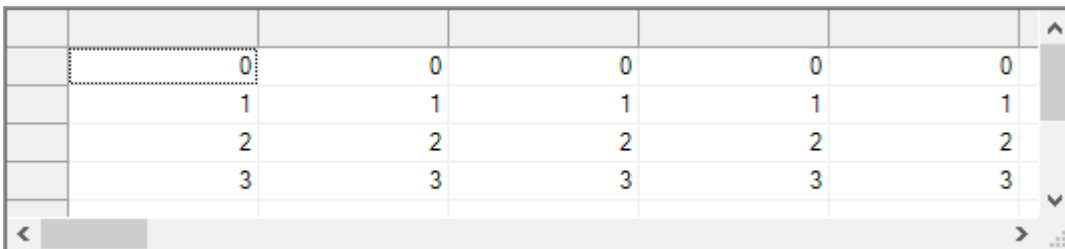
#### To write code in Visual Basic

```
Visual Basic  
C1FlexGrid1.Transpose ()
```


#### To write code in C#

```
C#  
c1FlexGrid1.Transpose ();
```

Run the project. The grid now displays the data as follows:



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

 Note that Transpose method only works in unbound mode. In addition, when you transpose the grid with a sorted column, the transpose method removes sorting before transposing. This means that the sort icon is not displayed after the transposing is done.

## Restricting Grid Editing

To disable editing of the entire grid, specific column or specific row, set the **AllowEditing** property to **False**. Conversely, to allow editing, set the **AllowEditing** property to **True**, which is the default value.

## Disable Editing for the Entire Grid

To disable editing for the entire grid, setting the **AllowEditing** property to **False** either in the designer or in code.

### In the Designer

In the **C1FlexGrid Tasks** menu, uncheck the **Enable Editing** check box.

Alternatively, locate the **AllowEditing** property in the Properties window and set it to **False**.

### In Code

Add the following code to the **Form\_Load** event:

#### To write code in Visual Basic

```
Visual Basic  
Me.C1FlexGrid1.AllowEditing = False
```

**To write code in C#**

```
C#  
this.c1FlexGrid1.AllowEditing = false;
```

## Disable Editing for a Specific Column

To disable editing for a specific column, set the [AllowEditing](#) property to **False** either in the designer or in code.

### In the Designer

1. Select the column in the grid that you would like to edit. This will open the **Column Tasks** menu for that column.
2. Uncheck the **Allow Editing** check box.

Alternatively, the **AllowEditing** property can be set using the **C1FlexGrid Column Editor**:

1. Open the **C1FlexGrid Column Editor**. For details on how to access the **C1FlexGrid Column Editor**, see [Accessing the C1FlexGrid Column Editor](#).
2. Select the column that you would like to edit from the right pane and set the **AllowEditing** property to **False** in the left pane.
3. Click **OK** to close the editor.

### In Code

Add the following code to the **Form\_Load** event to restrict editing the *AtomicMass* column:

**To write code in Visual Basic**

```
Visual Basic  
Me.C1FlexGrid1.Cols("AtomicMass").AllowEditing = False  
' Is the same as: Me.C1FlexGrid1.Cols(4).AllowEditing = False
```

**To write code in C#**

```
C#  
this.c1FlexGrid1.Cols["AtomicMass"].AllowEditing = false;  
// Is the same as: this.c1FlexGrid1.Cols[4].AllowEditing = false;
```

## Disable Editing for a Specific Row

To disable editing for a specific row, add the following code to the **Form\_Load** event to set the **AllowEditing** property to **False**. In this example, the code will restrict editing in the 6th row:

**To write code in Visual Basic**

```
Visual Basic  
Me.C1FlexGrid1.Rows(5).AllowEditing = False
```

**To write code in C#**

```
C#  
this.c1FlexGrid1.Rows[5].AllowEditing = false;
```

## Restricting Sorting for a Specific Column

To restrict sorting for a specific column, set the [AllowSorting](#) property to **False** either in the designer or in code.

### In the Designer

1. Select the column in the grid that you would like to edit. This will open the **Column Tasks** menu for that column.
2. Uncheck the **Allow Sorting** check box.

Alternatively, the `AllowSorting` property can be set using the **C1FlexGrid Column Editor**:

1. Open the **C1FlexGrid Column Editor**. For details on how to access the **C1FlexGrid Column Editor**, see [Accessing the C1FlexGrid Column Editor](#).
2. Select the column that you would like to edit from the right pane and set the `AllowSorting` property to **False** in the left pane.
3. Click **OK** to close the editor.

### In Code

Add the following code to the **Form\_Load** event to restrict sorting the *AtomicNumber* column:

**To write code in Visual Basic**

```
Visual Basic  
Me.C1FlexGrid1.Cols("AtomicNumber").AllowSorting = False  
' Is the same as: Me.C1FlexGrid1.Cols(1).AllowEditing = False
```

**To write code in C#**

```
C#  
this.c1FlexGrid1.Cols["AtomicNumber"].AllowSorting = false;  
// Is the same as: this.c1FlexGrid1.Cols[1].AllowEditing = false;
```

## Searching for Entries in a Column

To search for entries in a column as a user types, set the [AutoSearch](#) property to **FromCursor** to begin the search from the current row or **FromTop** to begin the search from the first scrollable row. Conversely, to disable the search, set the `AutoSearch` property to **None**, which is the default setting. This property can be set either in the designer or in code.

### In the Designer

Locate the **AutoSearch** property in the Properties window and set it to **FromTop**.

## In Code

Add the following code to the **Form\_Load** event to set the **AutoSearch** property to **FromTop**:

### To write code in Visual Basic

Visual Basic

```
Me.C1FlexGrid1.AutoSearch = C1.Win.C1FlexGrid.AutoSearchEnum.FromTop
```

### To write code in C#

C#

```
this.c1FlexGrid1.AutoSearch = C1.Win.C1FlexGrid.AutoSearchEnum.FromTop;
```

## This topic illustrates the following:

As the user types, the search will highlight the cell containing that letter. In this example, typing C in the *Element* column highlights Carbon.

AtomicNumber	Element	Symbol	AtomicMass	Group	Period	GroupName
1	Hydrogen	H	1.0079400	1	1	
2	Helium	He	4.0026020	18	1	Noble gas
3	Lithium	Li	6.9410000	1	2	Alkali metal
4	Beryllium	Be	9.0121820	2	2	Alkaline earth met
5	Boron	B	10.8110000	13	2	
6	Carbon	C	12.0107000	14	2	
7	Nitrogen	N	14.0067000	15	2	Pnictogen
8	Oxygen	O	15.9994000	16	2	Chalcogen
9	Flourine	F	18.9984032	17	2	Halogen
10	Neon	Ne	20.1797000	18	2	Noble gas
11	Sodium	Na	22.9897700	1	3	Alkali metal
12	Magnesium	Mg	24.3050000	2	3	Alkaline earth mel
13	Aluminium	Al	26.9815380	13	3	
14	Silicon	Si	28.0855000	14	3	

**Note:** If more than one entry begins with the same letter, typing the next letter will highlight the entry with those letters. For example, typing **He** in the *Element* column will highlight Helium.

## Searching for Entries in a Grid

FlexGrid's Search Panel provides you with a quick way to find data by searching the entire grid in one go. The panel searches and locates the relevant record in the millions of records present in the grid instantly.

### Search Panel Elements

The elements present in the search panel are as follows :

- **Search text box:** Allows you to enter text or string to be searched in the grid. The default watermark of the text box is Enter text to search. You can however change it by using the Watermark property of C1FlexGridSearchPanel.
- **Search button:** Allows you to obtain records that match the string entered in the search text box. You can choose to either display or hide the Search button within the search panel by using the ShowSearchButton property that accepts Boolean values. The default value of the property is True.
- **Clear button:** Allows to clear the string entered in the search text box. To display or hide the Clear button within the search panel, set the ShowClearButton property to True or False respectively. The default value of the property is True.

## Advanced Search Options

Search Panel offers advanced search options that enable you to modify the behavior of the panel while executing a search.

- **Search modes:** There are two modes of searching the grid, manually and automatically. To manually search the grid, you can simply click the Search button or press the ENTER key. While to automatically search, you just let the search panel search the grid on its own after a small delay. Set the SearchMode property to SearchClick or Always from the SearchMode enumeration for manual or automatic search, respectively.
- **Search delay:** You can set the delay (in milliseconds) after which automatic search is initiated by using the SearchDelay property. Notice that you need to set the SearchMode property to Always to be able to set the search delay.
- **Search highlighting:** To easily view searched records, you can highlight the search string within those records by setting the HighlightSearchResults property to True.

The following image displays search results obtained after executing a search through FlexGrid Search Panel.

The screenshot shows a search panel with a text box containing 'Louisiana'. Below the text box is a table with the following data:

ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	Unit
65	Louisiana Fiery Hot		2	2 32 - 8 oz bottles	21.05	
66	Louisiana Hot Spice		2	2 24 - 8 oz jars		17

To work with FlexGrid's Search Panel, create an instance of the C1FlexGridSearchPanel class and associate it with an instance of the C1FlexGrid class. You can use the SetC1FlexGridSearchPanel method that accepts two parameters, namely the C1FlexGrid and the C1FlexGridSearchPanel instance, to associate search panel with the grid. In addition, the default appearance and behavior of FlexGrid Search Panel can be customized by using the properties mentioned below.

The following code snippet customizes FlexGrid Search Panel and associates it with FlexGrid. This code uses the sample created in [Quick Start](#).

- **Visual Basic**

```
' Set the search mode
C1FlexGridSearchPanel1.SearchMode = C1.Win.C1FlexGrid.SearchMode.Always

' set the search delay when the search mode is automatic
C1FlexGridSearchPanel1.SearchDelay = 2
```

```
' set the watermark for the search text box
C1FlexGridSearchPanel1.Watermark = "Enter Search String"

' associate Search Panel with FlexGrid
C1FlexGridSearchPanel1.SetC1FlexGridSearchPanel(C1FlexGrid1, C1FlexGridSearchPanel1)

    • C#

// Set the search mode
c1FlexGridSearchPanel1.SearchMode = C1.Win.C1FlexGrid.SearchMode.Always;

// set the search delay when the search mode is automatic
c1FlexGridSearchPanel1.SearchDelay = 2;

// set the watermark for the search text box
c1FlexGridSearchPanel1.Watermark = "Enter Search String";

// associate Search Panel with FlexGrid
c1FlexGridSearchPanel1.SetC1FlexGridSearchPanel(c1FlexGrid1, c1FlexGridSearchPanel1);
```

## Setting a Cell's Value to Zero When Users Press the Delete Key

To set a cell's value to zero when a user presses the DELETE key, use **C1FlexGrid's** **KeyDown** event to catch when the DELETE key is pressed.

Add the following **KeyDown** event to your form:

### To write code in Visual Basic

#### Visual Basic

```
Private Sub C1FlexGrid1_KeyDown(ByVal sender As Object, ByVal e As
System.Windows.Forms.KeyEventArgs) Handles C1FlexGrid1.KeyDown
    If (e.KeyCode = Keys.Delete) Then
        C1FlexGrid1(C1FlexGrid1.Row, C1FlexGrid1.Col) = 0
    End If
End Sub
```

### To write code in C#

#### C#

```
private void c1FlexGrid1_KeyDown(object sender, System.Windows.Forms.KeyDownEventArgs
e)
{
    if (e.KeyCode == Keys.Delete)
    {
        c1FlexGrid1(c1FlexGrid1.Row, c1FlexGrid1.Col) = 0;
    }
}
```

## Setting Rows As Headers

To set rows as headers, set the [Caption](#) and [DataType](#) properties for each row.

1. Add the following code to the **Form\_Load** event to set the number of rows and columns to appear in the grid.

#### To write code in Visual Basic

##### Visual Basic

```
Me.C1FlexGrid1.Cols.Count = 5
Me.C1FlexGrid1.Rows.Count = 7
```

#### To write code in C#

##### C#

```
this.c1FlexGrid1.Cols.Count = 5;
this.c1FlexGrid1.Rows.Count = 7;
```

2. Add the rows to the RowCollection:

#### To write code in Visual Basic

##### Visual Basic

```
Dim row As C1.Win.C1FlexGrid.RowCollection = Me.C1FlexGrid1.Rows
```

#### To write code in C#

##### C#

```
C1.Win.C1FlexGrid.RowCollection row = this.c1FlexGrid1.Rows;
```

3. Set the Caption and DataType for each row.

#### To write code in Visual Basic

##### Visual Basic

```
row(1).Caption = "Date"
row(1).DataType = GetType(DateTime)

row(2).Caption = "Contact"
row(2).DataType = GetType(String)

row(3).Caption = "Phone"
row(3).DataType = GetType(String)
row(3).EditMask = "(999) 999-9999;*"

row(4).Caption = "Platform"
row(4).DataType = GetType(String)
row(4).ComboList = "|Windows XP|Windows 2000|Windows ME|Windows NT|Windows 98|Windows 95"

row(5).Caption = "Error Code"
row(5).DataType = GetType(Integer)

row(6).Caption = "Resolved"
```

```
row(6).DataType = GetType(Boolean)
```

#### To write code in C#

```
C#
row[1].Caption = "Date";
row[1].DataType = typeof(DateTime);

row[2].Caption = "Contact";
row[2].DataType = typeof(string);

row[3].Caption = "Phone";
row[3].DataType = typeof(string);
row[3].EditMask = "(999) 999-9999;*";

row[4].Caption = "Platform";
row[4].DataType = typeof(string);
row[4].ComboBox = "|Windows XP|Windows 2000|Windows ME|Windows NT|Windows
98|Windows 95";

row[5].Caption = "Error Code";
row[5].DataType = typeof(int);

row[6].Caption = "Resolved";
row[6].DataType = typeof(bool);
```

4. Format the headers to **Tahoma, 9pt, Bold** font.

#### To write code in Visual Basic

```
Visual Basic
Me.C1FlexGrid1.Styles("Fixed").Font = New Font("Tahoma", 9, FontStyle.Bold)
```

#### To write code in C#

```
C#
this.c1FlexGrid1.Styles["Fixed"].Font = new Font("Tahoma", 9, FontStyle.Bold);
```

5. Merge the fixed row and add a header to it:

#### To write code in Visual Basic

```
Visual Basic
Me.C1FlexGrid1.AllowMerging = C1.Win.C1FlexGrid.AllowMergingEnum.FixedOnly
row(0).AllowMerging = True
Dim rng As C1.Win.C1FlexGrid.CellRange = C1FlexGrid1.GetCellRange(0, 1, 0, 4)
rng.Data = "Call Log"
```

#### To write code in C#

```
C#
```



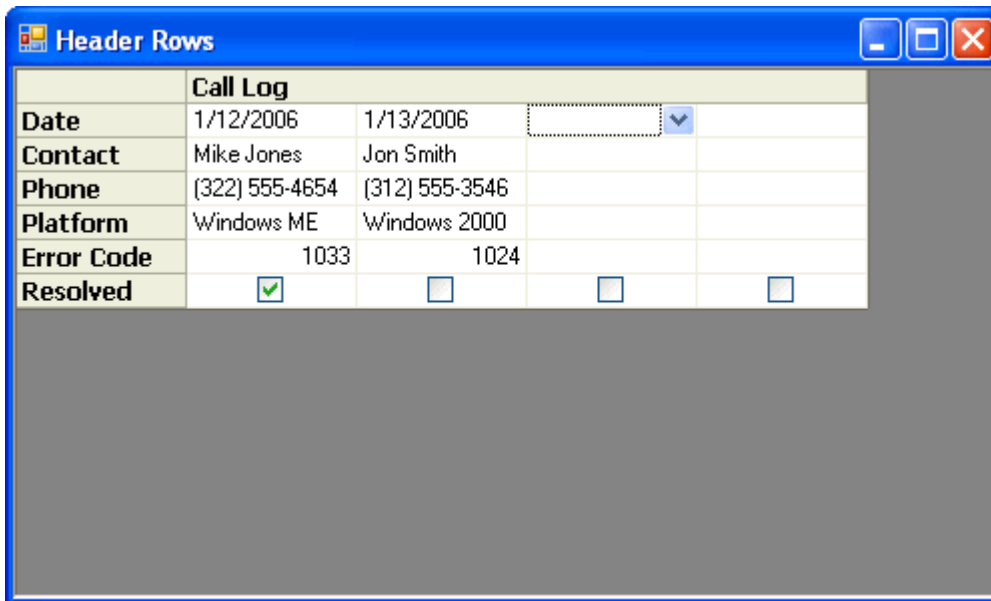
```

this.c1FlexGrid1.AllowMerging = C1.Win.C1FlexGrid.AllowMergingEnum.FixedOnly;
row[0].AllowMerging = true;
C1.Win.C1FlexGrid.CellRange rng = c1FlexGrid1.GetCellRange(0,1,0,4);
rng.Data = "Call Log";

```

## This topic illustrates the following:

The row headers appears in the first column and the each row will be formatted according to its DataType property.



	Call Log			
Date	1/12/2006	1/13/2006		
Contact	Mike Jones	Jon Smith		
Phone	(322) 555-4654	(312) 555-3546		
Platform	Windows ME	Windows 2000		
Error Code	1033	1024		
Resolved	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

## Setting the Font of a Single Cell

To set the font of a single cell, create a new style and assign it to a cell.

1. Create a new style.

### In the Designer

- Open the C1FlexGrid Style Editor. For details on how to access the C1FlexGrid Style Editor, see [Accessing the C1FlexGrid Style Editor](#).
- Click Add to create a new style.
- Double-click CustomStyle1, rename it myStyle, and press ENTER when finished.
- Do not exit the C1FlexGrid Style Editor.

### In Code

#### To write code in Visual Basic

Visual Basic

```

Dim cs As C1.Win.C1FlexGrid.CellStyle
cs = Me.C1FlexGrid1.Styles.Add("myStyle")

```

#### To write code in C#

```
C#
```

```
C1.Win.C1FlexGrid.CellStyle cs = this.c1FlexGrid1.Styles.Add("myStyle");
```

2. Set the font to **Tahoma, 10 pt, Bold**.

#### In the Designer

- In the **C1FlexGrid Style Editor**, locate the **Font** property in the right pane and click the **ellipsis** button. The **Font** dialog will appear.
- Set the **Font** box to **Tahoma**.
- Set the **Font style** box to **Bold**.
- Set the **Size** box to **10**.
- Click **OK** to close the **Font** dialog box, but do not exit the **C1FlexGrid Style Editor**.

#### In Code

Add the following code to the **Form\_Load** event:

##### To write code in Visual Basic

```
Visual Basic
```

```
cs.Font = New Font("Tahoma", 10, FontStyle.Bold)
```

##### To write code in C#

```
C#
```

```
cs.Font = new Font("Tahoma", 10, FontStyle.Bold);
```

3. Set the font color to **Blue**.

#### In the Designer

- In the **C1FlexGrid Style Editor**, locate the **ForeColor** property in the right pane and set it to **Blue**.
- Click **OK** to close the **C1FlexGrid Style Editor**.

#### In Code

Add the following code to the **Form\_Load** event:

##### To write code in Visual Basic

```
Visual Basic
```

```
cs.ForeColor = Color.Blue
```

##### To write code in C#

```
C#
```

```
cs.ForeColor = Color.Blue;
```

4. Assign the style to a cell by adding the following code to the **Form\_Load** event:

##### To write code in Visual Basic

```
Visual Basic
```

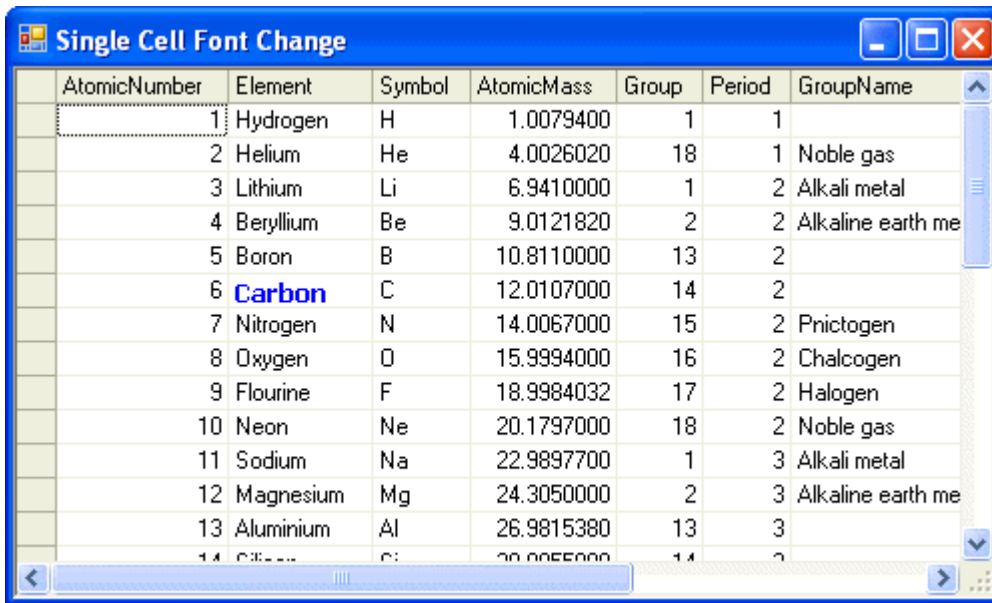
```
Me.C1FlexGrid1.SetCellStyle(6, 2, "myStyle")
```

## To write code in C#

```
C#
this.c1FlexGrid1.SetCellStyle(6, 2, "myStyle");
```

## This topic illustrates the following:

Carbon appears in Blue, Bold, 10 pt, Tahoma font.



## Setting the Text Delimiting Character in C1FlexGrid

To set the text delimiting character in `C1FlexGrid`, set the `Split` method.

Add the following code to the `Form_Load` event to set the delimiting character to a semicolon:

### To write code in Visual Basic

```
Visual Basic
Dim cols As String = "Product;Region;Salesperson;Sales;Bonus"
Dim colNames As String() = cols.Split(";")

Me.C1FlexGrid1.Cols.Count = 5
Me.C1FlexGrid1.Cols.Fixed = 0
Dim i%
For i = 0 To Me.C1FlexGrid1.Cols.Count - 1
    Me.C1FlexGrid1(0, i) = colNames(i)
    Me.C1FlexGrid1.Cols(i).Name = colNames(i)
Next
```

### To write code in C#

C#

```
string cols = "Product;Region;Salesperson;Sales;Bonus";
string[] colNames = cols.Split(new char[] { ';' });

this.c1FlexGrid1.Cols.Count = 5;
this.c1FlexGrid1.Cols.Fixed = 0;
for (int i = 0; i <= this.c1FlexGrid1.Cols.Count - 1; i++)
{
    this.c1FlexGrid1[0, i] = colNames[i];
    this.c1FlexGrid1.Cols[i].Name = colNames[i];
}
```

## This topic illustrates the following:

The string in the **Split** method determines the delimiting character. Changing the semicolon to a comma in both the **Split** method and the string will produce the same output as the semicolon.

## Sorting Multiple Columns

To sort multiple columns, set each column's [Sort](#) property and use the [Sort](#) method to sort according to the column settings.

1. Add the following code to the **Form\_Load** event to set the second column to sort in ascending order and the third column to sort in descending order.

### To write code in Visual Basic

Visual Basic

```
Me.C1FlexGrid1.Cols(1).Sort = C1.Win.C1FlexGrid.SortFlags.Ascending
Me.C1FlexGrid1.Cols(2).Sort = C1.Win.C1FlexGrid.SortFlags.Descending
```

### To write code in C#

C#

```
this.c1FlexGrid1.Cols[1].Sort = C1.Win.C1FlexGrid.SortFlags.Ascending;
this.c1FlexGrid1.Cols[2].Sort = C1.Win.C1FlexGrid.SortFlags.Descending;
```

2. Add the following Sort method to sort according to the settings in columns two and three.

### To write code in Visual Basic

Visual Basic

```
Me.C1FlexGrid1.Sort(C1.Win.C1FlexGrid.SortFlags.UseColSort, 1, 2)
```

### To write code in C#

C#

```
this.c1FlexGrid1.Sort(C1.Win.C1FlexGrid.SortFlags.UseColSort, 1, 2);
```

## This topic illustrates the following:

Your grid will look like the following with the second column sorted in ascending order, then the third column in descending order. In this example, Neon appears before Argon in the *Element* column since the grid is sorted first by the *StandardState* column in ascending order, then by the *Element* column in descending order.

Standard State	Element	Symbol	AtomicNumber	AtomicMass	Group	Period
Gas	Oxygen	O	8	15.9994000	16	2
Gas	Nitrogen	N	7	14.0067000	15	2
Gas	Neon	Ne	10	20.1797000	18	2
Gas	Krypton	Kr	36	83.7980000	18	4
Gas	Hydrogen	H	1	1.0079400	1	1
Gas	Helium	He	2	4.0026020	18	1
Gas	Flourine	F	9	18.9984032	17	2
Gas	Chlorine	Cl	17	35.4530000	17	3
Gas	Argon	Ar	18	39.9480000	18	3
Liquid	Bromine	Br	35	79.9040000	17	4
Solid	Zinc	Zn	30	65.4090000	12	4
Solid	Vanadium	V	23	50.9415000	5	4
Solid	Titanium	Ti	22	47.8670000	4	4
Solid	Carbon	C	12	12.0107000	14	2

## Styling and Appearance

### Setting the Background Color of Columns and Rows

To set the background color of columns and rows, create a new style and assign it to a column and row.

### Setting the Background Color of Columns

1. Create a new style for the column.

#### In the Designer

- o Open the **C1FlexGrid Style Editor**. For details on how to access the **C1FlexGrid Style Editor**, see [Accessing the C1FlexGrid Style Editor](#).
- o Click **Add** to create a new style.
- o Double-click **CustomStyle1**, rename it **ColumnColor**, and press ENTER when finished.
- o Do not exit the **C1FlexGrid Style Editor**.

#### In Code

Add the following code to the **Form\_Load** event:

#### To write code in Visual Basic

```
Visual Basic
Dim cc As C1.Win.C1FlexGrid.CellStyle
```

```
cc = Me.C1FlexGrid1.Styles.Add("ColumnColor")
```

**To write code in C#**

C#

```
C1.Win.C1FlexGrid.CellStyle cc = this.c1FlexGrid1.Styles.Add("ColumnColor");
```

2. Set the **BackColor** color to **CornSilk**.

**In the Designer**

- In the **C1FlexGrid Style Editor**, locate the **BackColor** property in the right pane and set it to **CornSilk**.
- Click **OK** to close the **C1FlexGrid Style Editor**.

**In Code**

Add the following code to the **Form\_Load** event:

**To write code in Visual Basic**

Visual Basic

```
cc.BackColor = Color.Cornsilk
```

**To write code in C#**

C#

```
cc.BackColor = Color.Cornsilk;
```

3. Assign the style to a column by adding the following code to the **Form\_Load** event:

**To write code in Visual Basic**

Visual Basic

```
Me.C1FlexGrid1.Cols(2).Style = Me.C1FlexGrid1.Styles("ColumnColor")
```

**To write code in C#**

C#

```
this.c1FlexGrid1.Cols[2].Style = this.c1FlexGrid1.Styles["ColumnColor"];
```

## This topic illustrates the following:

The background color of the *Element* column is set to **CornSilk**.

## Setting the Background Color of Rows

1. Create a new style for the row.

**In the Designer**

- Open the **C1FlexGrid Style Editor**. For details on how to access the **C1FlexGrid Style Editor**, see [Accessing the C1FlexGrid Style Editor](#).
- Click **Add** to create a new style.

- Double-click **CustomStyle1**, rename it **RowColor**, and press ENTER when finished.
- Do not exit the **C1FlexGrid Style Editor**.

#### In Code

Add the following code to the **Form\_Load** event:

##### To write code in Visual Basic

```
Visual Basic
Dim rs As C1.Win.C1FlexGrid.CellStyle
rs = Me.C1FlexGrid1.Styles.Add("RowColor")
```

##### To write code in C#

```
C#
C1.Win.C1FlexGrid.CellStyle rs = this.c1FlexGrid1.Styles.Add("RowColor");
```

2. Set the **BackColor** to **PowderBlue**.

#### In the Designer

- In the **C1FlexGrid Style Editor**, locate the **BackColor** property and set it to **PowderBlue**.
- Click **OK** to close the **C1FlexGrid Style Editor**.

#### In Code

##### To write code in Visual Basic

```
Visual Basic
rs.BackColor = Color.PowderBlue
```

##### To write code in C#

```
C#
rs.BackColor = Color.PowderBlue;
```

3. Assign the style to a row by adding the following code to the **Form\_Load** event:

##### To write code in Visual Basic

```
Visual Basic
Me.C1FlexGrid1.Rows(8).Style = Me.C1FlexGrid1.Styles("RowColor")
```

##### To write code in C#

```
C#
this.c1FlexGrid1.Rows[8].Style = this.c1FlexGrid1.Styles["RowColor"];
```

## This topic illustrates the following:

The background color of the row is set to **PowderBlue**. Notice how the column color takes precedence over the row color.

AtomicNumber	Element	Symbol	AtomicMass	Group	Period	GroupName
1	Hydrogen	H	1.0079400	1	1	
2	Helium	He	4.0026020	18	1	Noble gas
3	Lithium	Li	6.9410000	1	2	Alkali metal
4	Beryllium	Be	9.0121820	2	2	Alkaline earth me
5	Boron	B	10.8110000	13	2	
6	Carbon	C	12.0107000	14	2	
7	Nitrogen	N	14.0067000	15	2	Pnictogen
8	Oxygen	O	15.9994000	16	2	Chalcogen
9	Flourine	F	18.9984032	17	2	Halogen
10	Neon	Ne	20.1797000	18	2	Noble gas
11	Sodium	Na	22.9897700	1	3	Alkali metal
12	Magnesium	Mg	24.3050000	2	3	Alkaline earth me
13	Aluminium	Al	26.9815380	13	3	
14	Si-	Si	28.0855000	14	3	

## Undoing a Sort

To undo a sort in C1FlexGrid when the grid is bound to a DataTable, set the **DefaultView** property to null.

Add the following code to the **Button1\_Click** event:

### To write code in Visual Basic

#### Visual Basic

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
    Handles Button1.Click
        CustTable.DefaultView.Sort = ""
    End Sub
```

### To write code in C#

#### C#

```
private void Button1_Click(object sender, System.EventArgs e)
{
    CustTable.DefaultView.Sort = "";
}
```

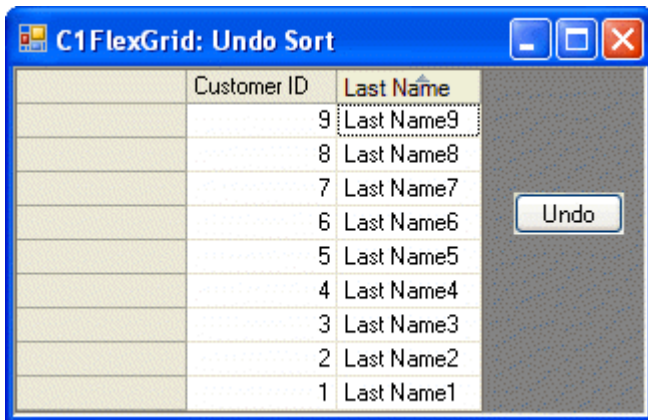


**Note:** The DataTable.**DefaultView** returns the DataView of the DataTable, and setting the sort string to null forces the DataView to undo the previous sort.

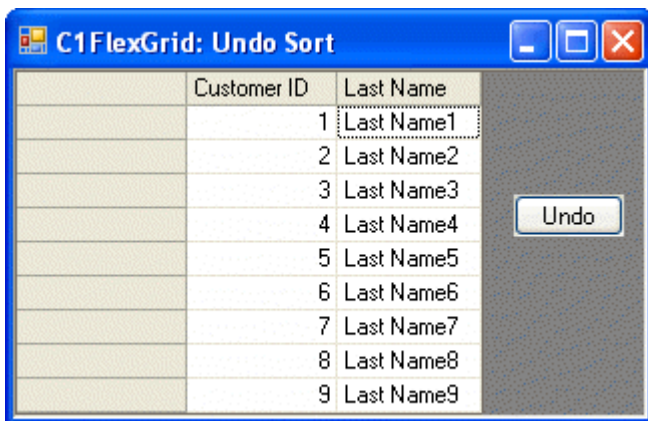
## This topic illustrates the following:

Click the *Last Name* column to sort on it.





Click the **Undo** button, and the sort will be undone.



## Using Password Entries in C1FlexGrid

To show placeholder characters (\*) in cells used for password entry, use the [SetupEditor](#) event.

1. Create a column for passwords in the grid and set the draw mode:

### To write code in Visual Basic

#### Visual Basic

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles MyBase.Load
    Me.C1FlexGrid1.Cols(0).Width = Me.C1FlexGrid1.Rows(0).HeightDisplay
    Me.C1FlexGrid1.ShowCursor = True
    Me.C1FlexGrid1.Cols(1).Caption = ((Me.C1FlexGrid1.Cols(1).Name) = "Password")
    Me.C1FlexGrid1.DrawMode = C1.Win.C1FlexGrid.DrawModeEnum.OwnerDraw
End Sub
```

### To write code in C#

#### C#

```
{
```

```
this.c1FlexGrid1.Cols[0].Width = this.c1FlexGrid1.Rows[0].HeightDisplay;
this.c1FlexGrid1.ShowCursor = true;
this.c1FlexGrid1.Cols[1].Caption = this.c1FlexGrid1.Cols[1].Name =
"Password";
this.c1FlexGrid1.DrawMode = C1.Win.C1FlexGrid.DrawModeEnum.OwnerDraw;
}
```

2. Add the following code for the **SetupEditor** event. This code will hide characters that are entered by the user.

#### To write code in Visual Basic

##### Visual Basic

```
Visual Basic
Private Sub C1FlexGrid1_SetupEditor(ByVal sender As Object, ByVal e As
C1.Win.C1FlexGrid.RowColEventArgs) Handles C1FlexGrid1.SetupEditor
    Dim tb As TextBox = Me.C1FlexGrid1.Editor
    If Not (tb Is Nothing) Then
        If Me.C1FlexGrid1.Cols(e.Col).Name = "Password" Then
            tb.PasswordChar = "*"c
        Else
            tb.PasswordChar = CChar(0)
        End If
    End If
End Sub
```

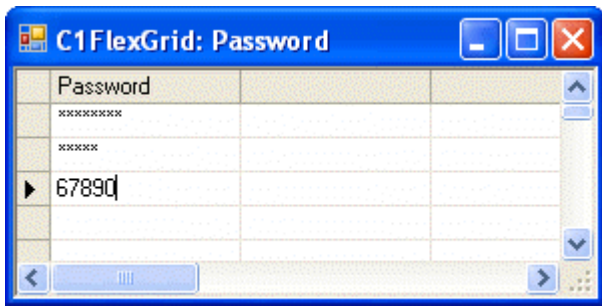
#### To write code in C#

##### C#

```
private void c1FlexGrid1_SetupEditor(object sender,
C1.Win.C1FlexGrid.RowColEventArgs e)
{
    TextBox tb = this.c1FlexGrid1.Editor as TextBox;
    if (tb != null)
    {
        if (this.c1FlexGrid1.Cols[e.Col].Name == "Password")
            tb.PasswordChar = '*';
        else
            tb.PasswordChar = (char)0;
    }
}
```

## This topic illustrates the following:

When the user enters a password in the *Password* column and presses ENTER, the text is automatically converted to asterisks.



## Hiding Characters Already Entered

To hide the characters that have already been entered and do not need edited, use the [OwnerDrawCell](#) event.

1. Add the following code for the **OwnerDrawCell** event. This code will hide the characters that have already been entered and do not need edited.

### To write code in Visual Basic

#### Visual Basic

```
Private Sub C1FlexGrid1_OwnerDrawCell(ByVal sender As Object, ByVal e As
C1.Win.C1FlexGrid.OwnerDrawCellEventArgs) Handles C1FlexGrid1.OwnerDrawCell
    If e.Row >= Me.C1FlexGrid1.Rows.Fixed And Me.C1FlexGrid1.Cols(e.Col).Name =
"Password" Then
        e.Text = New String("*"c, e.Text.Length)
    End If
End Sub
```

### To write code in C#

#### C#

```
private void c1FlexGrid1_OwnerDrawCell(object sender,
C1.Win.C1FlexGrid.OwnerDrawCellEventArgs e)
{
    if (e.Row >= this.c1FlexGrid1.Rows.Fixed &&
this.c1FlexGrid1.Cols[e.Col].Name == "Password")
    {
        e.Text = new string('*', e.Text.Length);
    }
}
```

2. Add the following code to the **Form\_Load** event:

### To write code in Visual Basic

#### Visual Basic

```
Me.C1FlexGrid1(1, 1) = "123456"
```

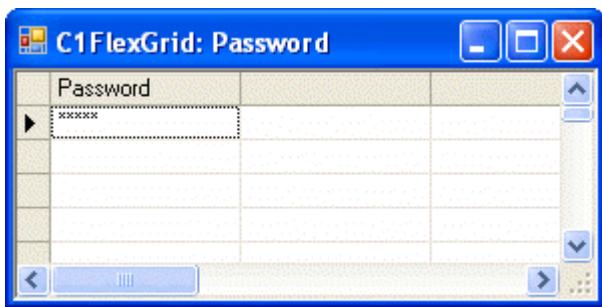
### To write code in C#

#### C#

```
this.c1FlexGrid1[1,1] = "123456";
```

## This topic illustrates the following:

Run the project again and notice the numbers are automatically loaded onto the form in the *Password* column as asterisks.



## Word Wrapping in a Header or Fixed Row

To set word wrapping in a header or fixed row, set the [Height](#) and [WordWrap](#) properties.

1. Set the [Caption](#) property for a column header in the grid.

### In the Designer

- Select a column in the grid. This will open the **Column Tasks** menu for that column.
- In the **Column Caption** box, enter **Word Wrapping in Header**.

Alternatively, the [Caption](#) property can also be set using the **C1FlexGrid Column Editor**.

- Open the **C1FlexGrid Column Editor**. For details on how to access the **C1FlexGrid Column Editor**, see [Accessing the C1FlexGrid Column Editor](#).
- Select a column in the right pane and set the **Caption** property in the left pane to **Word Wrapping in Header**.
- Click **OK** to close the editor.

### In Code

Add the following code to the **Form\_Load** event.

#### To write code in Visual Basic

Visual Basic

```
Me.C1FlexGrid1.Cols(1).Caption = "Word Wrapping in Header"
```

#### To write code in C#

C#

```
this.c1FlexGrid1.Cols[1].Caption = "Word Wrapping in Header";
```

2. Set the row height of the header.

#### To write code in Visual Basic

Visual Basic

```
Me.C1FlexGrid1.Rows(0).Height = 3 * Me.C1FlexGrid1.Rows.DefaultSize
```

### To write code in C#

C#

```
this.c1FlexGrid1.Rows[0].Height = 3 * this.c1FlexGrid1.Rows.DefaultSize;
```

3. Enable word wrapping for the fixed cells.

### In the Designer

- Open the **C1FlexGrid Style Editor**. For details on how to access the **C1FlexGrid Style Editor**, see [Accessing the C1FlexGrid Style Editor](#).
- Select **Fixed** in the **Built-In Styles** list.
- Locate the **WordWrap** property in the right pane and set it to **True**.
- Click **OK** to close the designer.

### In Code

Add the following code to the **Form\_Load** event:

### To write code in Visual Basic

Visual Basic

```
Me.C1FlexGrid1.Styles("Fixed").WordWrap = True
```

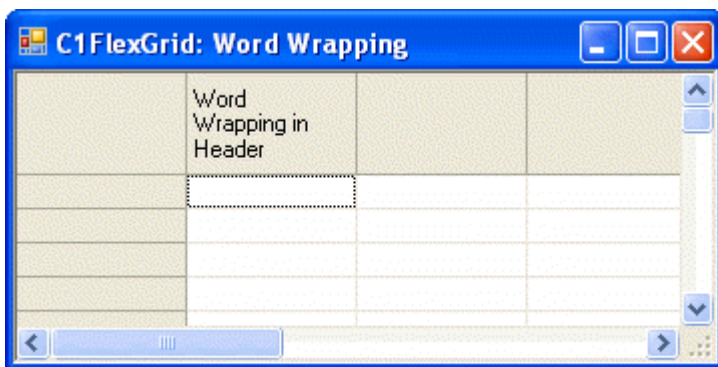
### To write code in C#

C#

```
this.c1FlexGrid1.Styles["Fixed"].WordWrap = true;
```

## This topic illustrates the following:

In this example, a three-line header is created, and the text is wrapped.



## FlexGrid for WinForms Top Tips

The following tips were compiled from frequently asked user questions posted in the [C1FlexGrid forum](#).

### Tip 1: Use the BeginUpdate/EndUpdate Methods to Increase Performance

Every time a cell value is changed, or rows and columns are added or removed from the grid, some calculations are performed in order to recalculate the grid layout and update the display.

You can increase performance significantly by enclosing the changes in calls to the **BeginUpdate** and **EndUpdate** methods. For example:

#### To write code in Visual Basic

```
Visual Basic
' call BeginUpdate before updating the grid.
_flex.BeginUpdate()

' Make the changes.
Try
    Dim r As Integer = 0
    Do While (r < _flex.Rows.Count)
        Dim c As Integer = 0
        Do While (c < _flex.Cols.Count)
            _flex(r, c) = (r + c)
            c += 1
        Loop
        r += 1
    Loop
Finally
    ' Always call EndUpdate when done.
    _flex.EndUpdate()
End Try
```


#### To write code in C#

```
C#
// call BeginUpdate before updating the grid.
_flex.BeginUpdate();

// Make the changes.
try
{
    for (int r = 0; r < _flex.Rows.Count; r++)
    {
        for (int c = 0; c < _flex.Cols.Count; c++)
        {
            _flex[r, c] = r + c;
        }
    }
}
```

```
        }  
    }  
}  
finally  
{  
  
    // Always call EndUpdate when done.  
    _flex.EndUpdate();  
}
```


Note the use of a **try/finally** block to ensure that the [C1FlexGridBase.EndUpdate](#) method is called even if the update code fails and throws an exception.

 **Note:** The **BeginUpdate** and **EndUpdate** methods were added in the 2010/v1 release of the **C1FlexGrid**. In previous versions, the **Redraw** property was used for the same purpose. This change was made to increase source compatibility with other controls that use the **BeginUpdate** and **EndUpdate** pattern.

## Tip 2: Use the AutoResize Property to Increase Performance

When a bound grid retrieves data from a data source, it measures every cell and sets the column widths so they fit all the data. This ensures the grid layout is optimal to display the data in the data source, but it can be time-consuming if the data source is large (say more than a few thousand rows).

In these cases, you should consider setting the [AutoResize](#) property to **False** and setting the column widths in code.

 **Note:** Starting with the 2010/v1 release of the **C1FlexGrid**, the **AutoResize** property defaults to **False**. If your data source contains a relatively small number of items and you want the grid to resize the columns automatically, you can either set the **AutoResize** property to **True** manually or simply call the **AutoSizeCols** methods after binding or populating the grid. This change was made to increase performance in cases where the grid is bound to large data sources.

## Tip 3: Use the DrawMode Property to Assign Styles Dynamically Based on Cell Values

The grid allows you to create cell styles and assign them to rows, columns, and arbitrary cell ranges. You can use this capability to format cells based on their values. For example, cells that contain negative values can display them in red.

You can do this by assigning styles to cells with the [SetCellStyle](#) method, but in this case you have to update the style whenever the cell value changes. Also, if the grid is bound to a data source, styles will be lost whenever the data source is reset (after sorting and filtering operations for example).

A better alternative in these cases is to use the grid's **OwnerDraw** feature and select styles dynamically, based on the cell values. For example, the code below shows negative values in red and values above 1,000 in green:

### To write code in Visual Basic

```
Visual Basic  
  
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs)  
    Handles MyBase.Load  
  
    ' Fill a column with random values.  
    _flex.Cols(1).DataType = GetType(Integer)
```

```

Dim rnd As New Random
Dim r As Integer = 1
Do While (r < _flex.Rows.Count)
    _flex(r, 1) = rnd.Next(-10000, 10000)
    r += 1
Loop

' Create style used to show negative values.
_flex.Styles.Add("Red").ForeColor = Color.Red

' Create style used to show values >= 1000.
_flex.Styles.Add("Green").ForeColor = Color.Green

' Enable OwnerDraw by setting the DrawMode property.
_flex.DrawMode = C1.Win.C1FlexGrid.DrawModeEnum.OwnerDraw
End Sub

Private Sub _flex_OwnerDrawCell(ByVal sender As Object, ByVal e As
C1.Win.C1FlexGrid.OwnerDrawCellEventArgs) Handles _flex.OwnerDrawCell

    ' Check that the row and column contain integer data.
    If ((e.Row > 0) AndAlso (_flex.Cols(e.Col).DataType Is GetType(Integer))) Then

        ' Get value in cell about to be painted.
        Dim value As Integer = CInt(_flex(e.Row, e.Col))
        If (value < 0) Then

            ' If the Cell value < 0, use the Red style.
            e.Style = _flex.Styles("Red")
        ElseIf (value >= 1000) Then

            ' If the Cell value >= 1000, use the Green style.
            e.Style = _flex.Styles("Green")
        End If
    End If
End Sub

```

**To write code in C#**

```

C#
private void Form1_Load(object sender, EventArgs e)
{

    // Fill a column with random values.
    _flex.Cols[1].DataType = typeof(int);
    Random rnd = new Random();
    for (int r = 1; r < _flex.Rows.Count; r++)
    {
        _flex[r, 1] = rnd.Next(-10000, 10000);
    }
}

```



```
// Create style used to show negative values.
_flex.Styles.Add("Red").ForeColor = Color.Red;

// Create style used to show values >= 1000.
_flex.Styles.Add("Green").ForeColor = Color.Green;

// Enable OwnerDraw by setting the DrawMode property.
_flex.DrawMode = C1.Win.C1FlexGrid.DrawModeEnum.OwnerDraw;
_flex.OwnerDrawCell += new
C1.Win.C1FlexGrid.OwnerDrawCellEventHandler(_flex_OwnerDrawCell);
}

private void _flex_OwnerDrawCell(object sender,
C1.Win.C1FlexGrid.OwnerDrawCellEventArgs e)
{

    // Check that the row and column contain integer data.
    if (e.Row > 0 && _flex.Cols[e.Col].DataType == typeof(int))
    {

        // Get value in cell about to be painted.
        int value = (int)_flex[e.Row, e.Col];
        if (value < 0)
        {

            // If the Cell value < 0, use the Red style.
            e.Style = _flex.Styles["Red"];
        }
        else if (value >= 1000)
        {

            // If the Cell value >= 1000, use the Green style.
            e.Style = _flex.Styles["Green"];
        }
    }
}
```

## Tip 4: Do Not Modify Styles in the OwnerDrawCell Event

Note that the code in Tip 3 does not modify the [CellStyle](#) object passed as a parameter in the [OwnerDrawCell](#) event. Instead, it assigns a new value to the **e.Style** parameter.

That is important because the [CellStyle](#) passed to the event handler is often used by other cells. For example, you could, unintentionally change the grid's normal style, which would affect most cells in the grid.

The examples below illustrate the difference:

### To write code in Visual Basic

#### Visual Basic

```
' ** CORRECT APPROACH:
Private Sub _flex_OwnerDrawCell(ByVal sender As Object, ByVal e As
```

```
C1.Win.C1FlexGrid.OwnerDrawCellEventArgs) Handles _flex.OwnerDrawCell

    ' Select style to use when painting this cell:
    e.Style = MyStyleSelector(e.Row, e.Col)
End Sub
```

### To write code in C#

```
C#

// ** CORRECT APPROACH:
private void _flex_OwnerDrawCell(object sender,
C1.Win.C1FlexGrid.OwnerDrawCellEventArgs e)
{

    // Select style to use when painting this cell:
    e.Style = MyStyleSelector(e.Row, e.Col);
}
```

Contrast this with the following:

### To write code in Visual Basic

```
Visual Basic

' ** WRONG APPROACH:
Private Sub _flex_OwnerDrawCell(ByVal sender As Object, ByVal e As
C1.Win.C1FlexGrid.OwnerDrawCellEventArgs) Handles _flex.OwnerDrawCell

    ' Select style to use when painting this cell:
    ' This is bad because changing any DataGridViewCellStyle objects invalidates the
    ' grid, which would cause this event handler to be called over and
    ' over again.
    e.Style.Color = MyColorSelector(e.Row, e.Col)
End Sub
```

### To write code in C#

```
C#

// ** WRONG APPROACH:
private void _flex_OwnerDrawCell(object sender,
C1.Win.C1FlexGrid.OwnerDrawCellEventArgs e)
{

    // Select style to use when painting this cell:
    // This is bad because changing any DataGridViewCellStyle objects invalidates the
    // grid, which would cause this event handler to be called over and
    // over again.
    e.Style.Color = MyColorSelector(e.Row, e.Col);
}
```

## Tip 5: Use the Trimming Property to Show Ellipses in a Single Column of the Grid

The [Trimming](#) property should be used to show ellipses in a single column of the grid. To determine how long strings are trimmed to fit the cell, the Trimming property can be set to either **None**, **Character**, **Word**, **EllipsisCharacter**, **EllipsisWord**, or **EllipsisPath**.

The following table describes each of the be trimming options:

Member Name	Description
<b>Character</b>	Specifies that the text is trimmed to the nearest character.
<b>EllipsisCharacter</b>	Specifies that the text is trimmed to the nearest character, and an ellipsis is inserted at the end of a trimmed line.
<b>EllipsisPath</b>	The center is removed from the trimmed lines and replaced by an ellipsis. The algorithm keeps as much of the last slash-delimited segment of the line as possible.
<b>EllipsisWord</b>	Specifies that the text is trimmed to the nearest word and an ellipsis is inserted at the end of a trimmed line.
<b>None</b>	Specifies no trimming.
<b>Word</b>	Specifies that the text is trimmed to the nearest word.

The following code sets the Trimming property to show ellipses at the end of the second column, with the text trimmed to the nearest character:

#### To write code in Visual Basic

Visual Basic

```
_flex.Cols(1).StyleNew.Trimming = StringTrimming.EllipsisCharacter
```

#### To write code in C#

C#

```
_flex.Cols[1].StyleNew.Trimming = StringTrimming.EllipsisCharacter;
```

## Tip 6: Use the WordWrap Property to Show Multiple Line Text in a Cell

When showing multiple lines of text in a cell, use the [WordWrap](#) and [Height](#) properties. The WordWrap property determines whether the grid should automatically break long strings that contain spaces and display them in multiple lines. Strings that contain hard line breaks (vbCrLf or "\n\r") are always displayed in multiple lines.

Multiple line text can be displayed in both fixed and scrollable cells. For an example setting multiple line text in a fixed cell, see [Word Wrapping in a Header or Fixed Row](#).

The following code sets display multiple line text in a scrollable cell:

#### To write code in Visual Basic

## Visual Basic

```
' Set the WordWrap property.
_flex.Styles("Normal").WordWrap = True

' Set the row height.
_flex.Rows(1).Height = 2 * fg.Rows.DefaultSize

' Add text to the cell.
_flex(1, 2) = "This is the first line." & ControlChars.CrLf & " This is the second
line."
```

## To write code in C#

## C#

```
// Set the WordWrap property.
_flex.Styles["Normal"].WordWrap = true;

// Set the row height.
_flex.Rows[1].Height = 2 * fg.Rows.DefaultSize;

// Add text to the cell.
_flex[1, 2] = "This is the first line. \r\n This is the second line.";
```

## Tip 7: Use the Sort Property to Retrieve Data Sorting when Bound to a DataTable

If the grid is bound to a `DataTable`, the user can maintain the way the grid is sorted when data is refreshed. This can be achieved by using the default view's **Sort** property and a sort expression.

The **Sort** property uses a string containing the column name followed by **ASC** (to sort the column in ascending order) or **DESC** (to sort the column in descending order). By default, columns are sorted in ascending order. Multiple columns can be sorted by entering each column name separated by a comma.

A sort expression can include names of grid columns or a calculation. Setting the sort expression at run time immediately reflects the changes in the data view.

## To write code in Visual Basic

## Visual Basic

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles MyBase.Load
    Me.ProductsTableAdapter.Fill(Me.NwindDataSet.Products)
End Sub

Private Sub btn_Sort_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btn_Sort.Click

    ' Sort the data by the UnitsInStock column then by the ProductID column.
    Me.ProductsBindingSource.Sort = "UnitsInStock ASC, ProductID ASC"
End Sub
```

```
Private Sub btn_ClearSort_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btn_ClearSort.Click

    ' Clear the sort.
    Me.ProductsBindingSource.Sort = ""
End Sub
```

**To write code in C#**

```
C#
private void Form1_Load(object sender, System.EventArgs e)
{
    this.productsTableAdapter.Fill(this.nwindDataSet.Products);
}

private void btn_Sort_Click(object sender, EventArgs e)
{
    // Sort the data by the UnitsInStock column then by the ProductID column.
    this.productsBindingSource.Sort = "UnitsInStock ASC, ProductID ASC";
}

private void btn_ClearSort_Click(object sender, EventArgs e)
{
    // Clear the sort.
    this.productsBindingSource.Sort = "";
}
```

**Tip 8: Control the Number of Characters to be Entered in a Column with the SetupEditor Event**

To set the maximum number of characters a user can enter for any given column use the [SetupEditor](#) event. For this example a **C1TextBox** has been set as the editor for **C1FlexGrid**. You must declare an external editor to be used with **C1FlexGrid** in the [StartEdit](#) event:

**To write code in Visual Basic**

```
Visual Basic
Private Sub _flex_StartEdit(ByVal sender As System.Object, ByVal e As
C1.Win.C1FlexGrid.RowColEventArgs) Handles _flex.StartEdit
    _flex.Editor = C1TextBox
End Sub
```

**To write code in C#**

```
C#
private void _flex_StartEdit(object sender, C1.Win.C1FlexGrid.RowColEventArgs e)
{
    _flex.Editor = c1TextBox;
}
```

Now that you have the grid's editor set up you can use the following code to allow 20 characters to be entered in the 3rd column and only 10 characters to be entered in the rest of the grid's columns (remember the following code must be entered in the [SetupEditor](#) event):

## To write code in Visual Basic

### Visual Basic

```
Private Sub _flex_SetupEditor(ByVal sender As Object, ByVal e As
C1.Win.C1FlexGrid.RowColEventArgs) Handles _flex.SetupEditor

    ' Set the 3rd column to allow 20 characters and the rest only 10.
    If e.Col = 2 Then
        CType(fg.Editor, C1TextBox).MaxLength = 20
    Else
        CType(fg.Editor, C1TextBox).MaxLength = 10
    End If
End Sub
```

## To write code in C#

### C#

```
private void _flex_SetupEditor(object sender, RowColEventArgs e)
{

    // Set the 3rd column to allow 20 characters and the rest only 10.
    if (e.Col == 2)
        c1TextBox.MaxLength = 20;
    else
        c1TextBox.MaxLength = 10;
}
```

## C1FlexGridClassic Control

**C1FlexGridClassic** is a control that derives from [C1FlexGrid](#) and provides an object model that is virtually identical to the **VSFlexGrid** ActiveX control. C1FlexGridClassic was developed to allow easy migration of existing **VSFlexGrid** projects.

The source code for C1FlexGridClassic is provided as a sample. You can use it as a reference that shows how to use the **C1FlexGrid** control as a base class in the development of custom grid controls.