ComponentOne

FlexGrid for Windows Phone

ComponentOne, a division of GrapeCity

201 South Highland Avenue, Third Floor

Pittsburgh, PA 15206 • USA

Internet: info@ComponentOne.com

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

Sales

E-mail: sales@componentone.com

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

This manual was produced using ComponentOne Doc-To-Help™.

Table of Contents

ComponentOne FlexGrid for Windows Phone Overview

Experience the FlexGrid's legendary flexibility, speed, and small footprint on the Windows Phone! Display tabular data across columns and down rows with this simple, yet powerful grid control. **ComponentOne FlexGrid® for Windows Phone** supports sorting, cell merging and even cell editing with a platform specific UI.

Help with ComponentOne Studio for Windows Phone

Getting Started

For information on installing **ComponentOne Studio for Windows Phone**, licensing, technical support, namespaces and creating a project with the control, please visit Getting Started with Studio for Windows Phone.

What's New

For a list of the latest features added to **ComponentOne Studio for Windows Phone**, visit What's New in Studio for Windows Phone.

FlexGrid for Windows Phone Assemblies and Controls

The **C1.Phone.FlexGrid.dll** assembly contains the **C1FlexGrid** control, a DataGrid control with a lightweight, flexible object model.

Main Classes

The following main classes are included in the C1.Phone.FlexGrid.dll assembly:

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

FlexGrid for Windows Phone Key Features

Make the most of **FlexGrid for Windows Phone** by taking advantage of the following key features:

- **Touch Interaction**

  The **FlexGrid for Windows Phone** has been remodeled to provide a user experience more fitting for the mobile world. Users can tap and flick the grid to scroll across columns, and flick or slide up and down to traverse rows. The data flies by smoothly to provide a clean and crisp experience.

- **Column Sorting**

  As expected with any control that calls itself a data grid, you can effortlessly sort data per column. This functionality is exposed through a simple method you can wire up to your application bar or any event.

- **Cell Editing**

  C1FlexGrid supports cell editing through a provided separate interface. The editing is not inline as it would be in a Web or desktop application, but rather in a hassle free pop-up perfect for the Windows Phone.

- **Flexible Data Binding**

  Easily bind the C1FlexGrid control to anything that implements **IEnumerable** or bind directly to a list of objects.

- **Unbound Mode**

  In addition to data binding support, C1FlexGrid also works great in unbound mode. Simply add rows and columns to the grid using familiar syntax from the WinForms version of the control. When unbound, the C1FlexGrid control stores data internally.

- **Column and Row Freezing**

  Freeze any number of rows and columns by simply setting the **Rows.Frozen** or **Columns.Frozen** properties. Frozen panes never scroll out of view.

- **Custom Cells**

  C1FlexGrid makes it easier to define custom cells by providing a more simpler interface, ICellFactory. Custom cells are useful if you want to display anything other than text in a grid cell.

Using FlexGrid for Windows Phone

This document introduces the **C1FlexGrid** control for Windows Phone.

Creating the C1FlexGrid

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

```
<phone:PhoneApplicationPage

..
  xmlns:my="clr-namespace:C1.Phone.FlexGrid;assembly=C1.Phone.FlexGrid"
>
<Grid x:Name="LayoutRoot">
  <my:C1FlexGrid/>
</Grid>
</UserControl>
```

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

```
var fg = new C1.Phone.FlexGrid.C1FlexGrid();
LayoutRoot.Children.Add(fg);
```

Populating the grid

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

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

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

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

The **editing** and **sorting & filtering** samples in the **ControlExplorer** sample installed with this product demonstrate how to populate the **C1FlexGrid**. The **ControlExplorer** sample is located in the *C:\Documents and Settings\<username>\My Documents\ComponentOne Samples\Studio for Windows Phone* or *C:\Users\<username>\Documents\ComponentOne Samples\Studio for Windows Phone* folder.

In these samples, the grid is bound to a customer list using the following code:

```
            // create the data
            var data = Customer.GetCustomerList(100);
            flexgrid.ItemsSource = data;

            // hide read-only "Country" column
            var col = flexgrid.Columns["Country"];
            col.Visible = false;
            // map countryID column so it shows country names instead
of their IDs
            Dictionary<int, string> dct = new Dictionary<int,
string>();
            foreach (var country in Customer.GetCountries())
            {
                dct[dct.Count] = country;
            }
            col = flexgrid.Columns["CountryID"];
            col.ValueConverter = new ColumnValueConverter(dct);
            col.HorizontalAlignment = HorizontalAlignment.Left;
            col.Header = "Country";
            col.Width = new GridLength(140);

            // provide auto-complete lists for first and last name
columns
            col = flexgrid.Columns["First"];
            col.ValueConverter = new
ColumnValueConverter(Customer.GetFirstNames(), false);
            col = flexgrid.Columns["Last"];
            col.ValueConverter = new
ColumnValueConverter(Customer.GetLastNames(), false);

            col = flexgrid.Columns["Hired"];
            col.Width = new GridLength(140);

            col = flexgrid.Columns["Weight"];
            col.Width = new GridLength(100);

            col = flexgrid.Columns["Active"];
            col.Width = new GridLength(100);
```

Sorting Data

C1FlexGrid supports sorting data directly using code. The following example code was taken from the **sorting & filtering** sample in the **ControlExplorer** installed with this product.

Note: The **ControlExplorer** sample is located in the *C:\Documents and Settings\<username>\My Documents\ComponentOne Samples\Studio for Windows Phone* or *C:\Users\<username>\Documents\ComponentOne Samples\Studio for Windows Phone* folder.

In this example, CollectionView **ICollectionView.SortDescriptions** are used to change the sort direction when the column header is tapped.

```
void flexgrid_Tap(object sender, C1GestureEventArgs e)
        {
            // detect tap and sort only if a column header was tapped
            var testInfo = flexgrid.HitTest(e);
            if (testInfo.CellType == CellType.ColumnHeader)
            {
                // get current column header property name

SortGrid(flexgrid.Columns[testInfo.Column].BoundPropertyName);
            }
        }
```

```
   void SortGrid(string propertyName)
```

```
        {
            // If we have a sort description with the current property
path, then we remove it from the collection and
            // add a new one with the opposite sort direction (if we
tap a column header twice, the sort direction changes).
            // If we don´t have a sort description with the current
property path, then we add it with ascending sort direction.
            var sortDescriptions =
flexgrid.CollectionView.SortDescriptions;

            using (flexgrid.CollectionView.DeferRefresh())
            {
                var sortDescription =
sortDescriptions.FirstOrDefault(sd =>
sd.PropertyName.Equals(propertyName));

                // if multiple selection is disabled, we always clear
the collection
                if (!_allowMultipleSort)
                {
                    sortDescriptions.Clear();
                }

                if (string.IsNullOrEmpty(sortDescription.PropertyName))
                {
                    sortDescription.PropertyName = propertyName;
                    sortDescription.Direction =
ListSortDirection.Ascending;
                    sortDescriptions.Add(sortDescription);
                }
                else
                {
                    sortDescriptions.Remove(sortDescription);
                    sortDescriptions.Add(new
SortDescription(propertyName, sortDescription.Direction ==
ListSortDirection.Ascending ? ListSortDirection.Descending :
ListSortDirection.Ascending));
                }
            }
        }
```

You can disable grid sorting by setting the **AllowSorting** property to false, or disable it for specific columns by setting the **Column.AllowSorting** property to false.

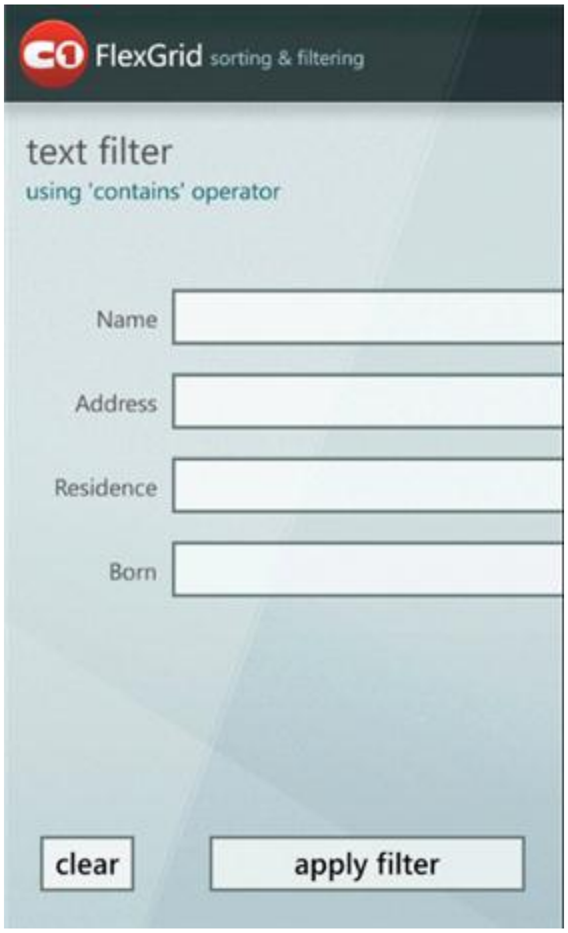
Filtering Data

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

In the **FlexGrid sorting & filtering** sample included with this product, the user can tap the **filter** button and a text filter popup appears.

```
 void btnFilter_Click(object sender, EventArgs e)
        {
            if (filterPopup.Content == null)
            {
                filterPopup.Content = new FlexGridFilter(flexgrid,
filterPopup);
            }
            var page =
C1.Phone.Extensions.FindParent<PhoneApplicationPage>(this);
            page.ApplicationBar.IsVisible = false;

            flexgrid.Visibility = System.Windows.Visibility.Collapsed;
            filterPopup.IsOpen = true;

        }
```

The text popup looks similar to this image:



When the user taps the **apply filter** button, the filter predicate is used to apply the filter. Here is the code used for the predicate:

```
private void Filter(object sender, RoutedEventArgs e)
        {
                _flexGrid.CollectionView.Filter = FilterPredicate;
                _parentPopup.IsOpen = false;
         }

        // predicate used to filter items in CollectionView
        private bool FilterPredicate(object item)
        {
            foreach (var colFilter in _currentFilters)
            {
                string filterValue = colFilter.Value.Trim().ToUpper();
                if ((colFilter.Column.PropertyInfo != null) &&
!string.IsNullOrEmpty(filterValue))
                {
                    string value =
colFilter.Column.PropertyInfo.GetValue(item, null) as string;
                    // use String.Contains method for filter sample
                    if (value == null ||
!value.ToUpper().Contains(filterValue))
                    {
                        return false;
                    }
                }
            }
            return true;
        }
    }
```

You can find the **sorting & filtering** sample in the **ControlExplorer** located in the *C:\Documents and Settings\<username>\My Documents\ComponentOne Samples\Studio for Windows Phone* or *C:\Users\<username>\Documents\ComponentOne Samples\Studio for Windows Phone* folder.

Unbound Mode

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

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

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

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

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

The indexing notation works in bound and unbound modes. In bound mode, the data is retrieved or applied to the items in the data source. In unbound mode, the data is stored internally by the grid.

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

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

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

The diagram below shows the new cell indexing scheme used in the **Windows Phone** version of the grid:

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

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

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

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

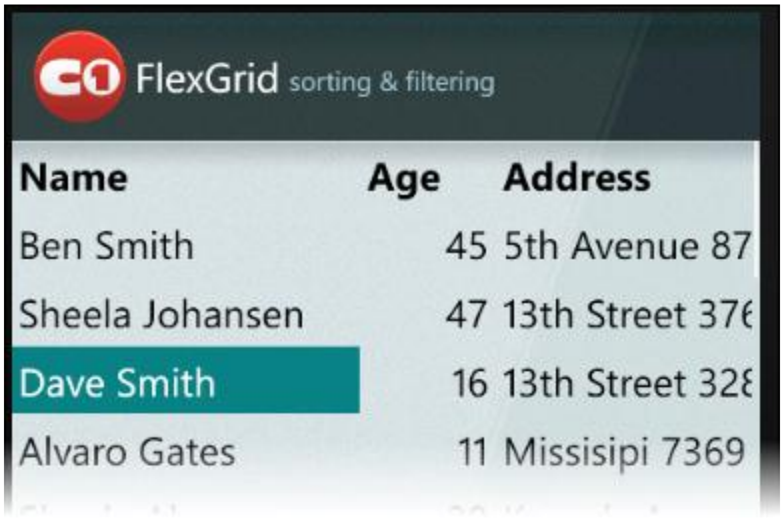
For example, you could use this code to customize the row headers:

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

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

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

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

Selection

The **C1FlexGrid** allows users to select an individual cell simply by tapping the cell. Currently, **FlexGrid for Windows Phone** supports **Cell** mode of the SelectionMode property. In **Cell** mode, the selection corresponds to a single cell.



Name	Age	Address
Ben Smith	45	5th Avenue 87
Sheela Johansen	47	13th Street 37
Dave Smith	16	13th Street 32
Alvaro Gates	11	Missisipi 7369

Custom Cells

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

Custom Cells in code: CellFactory class

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

The **ICellFactory** interface is very simple:

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

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

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

The first method, **CreateCell**, is responsible for creating **FrameworkElement** objects used to represent cells. The parameters include the grid that owns the cells, the type of cell to create, and the **CellRange** to be represented by the cells. The **CellType** parameter specifies whether the cell being created is a regular data cell, a row or column header, or the fixed cells at the top left and bottom right of the grid.

The second method, **CreateCellEditor**, is analogous to the first but creates a cell in edit mode.

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

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

Implementing custom **ICellFactory** classes is fairly easy because you can inherit from the default **CellFactory** class included with the **C1FlexGrid**. The default **CellFactory** class was designed to be class was designed to be extensible, so you can let it handle all the details of cell creation and customize only what you need.

The following section describes an example of grids that use custom **ICellFactory** classes used to implement functionality we hope you will find interesting.

Custom Cells in XAML: CellTemplate

If you prefer to create custom cells in XAML instead of writing code, you can do that as well. The **C1FlexGrid Column** object has a **CellTemplate** property that you can use to specify the visual elements responsible for showing and editing cells in the column.

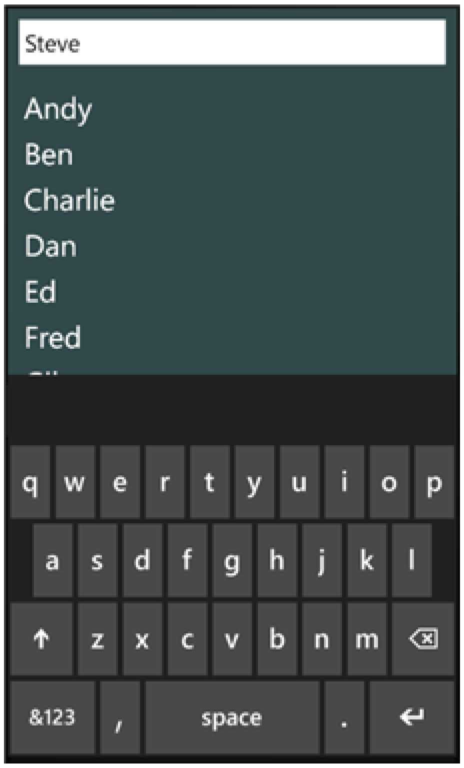
For example, the XAML code below defines custom visual elements used to show and edit values in a column. Cells in that column are shown as green, bold, center-aligned text, and edited using a textbox that has an edit icon next to it:

```
<c1:C1FlexGrid x:Name="_fgTemplated">

  <c1:C1FlexGrid.Columns>

    <!-- add a templated column -->
    <c1:Column ColumnName="_colTemplated" Header="Template"
Width="200">

      <!-- template for cells in display mode -->
      <c1:Column.CellTemplate>
        <DataTemplate>
          <TextBlock Text="{Binding Name}"
           Foreground="Green" FontWeight="Bold"
           VerticalAlignment="Center"/>
        </DataTemplate>
      </c1:Column.CellTemplate>

                </c1:Column>
  </c1:C1FlexGrid.Columns>
</c1:C1FlexGrid>
```

Editing Features

Editing is enabled by default. To disable editing at the grid level, set the C1FlexGrid.IsReadOnly property to true.

To edit a cell, simply double-tap the cell. An editing window pops up so you can make your changes. This applies to any cell type except for checkboxes. Simply tap a checkbox once to select or deselect it.



Data-Mapped columns

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

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

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

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

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

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

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

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

Using Custom Editors

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

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

- Implement a custom **CellFactory** class and override the **CreateCellEditor** method to create and bind your editor to the underlying data value.

Configuring Editors

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

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

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

Frozen Rows and Columns

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

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

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

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

When the user checks the **_chkFreezePanes** checkbox, the event handler sets the **Rows.Frozen** and **Columns.Frozen** properties to keep the rows and columns above and to the left of the current selection always in view.