10Tec iGrid.NET Control

version 14.0

Developer's Manual

10tec.com

CONTENTS

1. INTRODUCTION

1.1. Overview of iGrid.NET

The iGrid control is a cell matrix control. Mainly iGrid displays and operates on tabular data, but in addition it provides you with such features as sorting/grouping/filtering with various criteria. You can also easily define special sections like frozen areas for columns and/or rows and footer section with customizable grid totals.

One of the main distinctive features of iGrid is the ability to use styles to format its cells. The iGrid cell style object allows you to define the look and behavior of a cell and apply this style to a wide range of cells. Having such a cell style, you can quickly change the formatting parameters of all the cells this style is applied to even if you have hundreds of thousands of cells in your grid. This works similar to text styles you can use in text editors like Microsoft Word.

In addition to the standard set of grid control features, iGrid provides some unique features for customizing. For example, you can add additional buttons with predefined or custom actions to its scroll bars, or you can make the scroll bars always visible/hidden or semi-transparent. As for control appearance, iGrid can be rendered using one of the predefined rendering styles (the OS visual styles, classic 3D, or flat look), or you can completely redefine the drawing of its constituent elements to apply your own style.

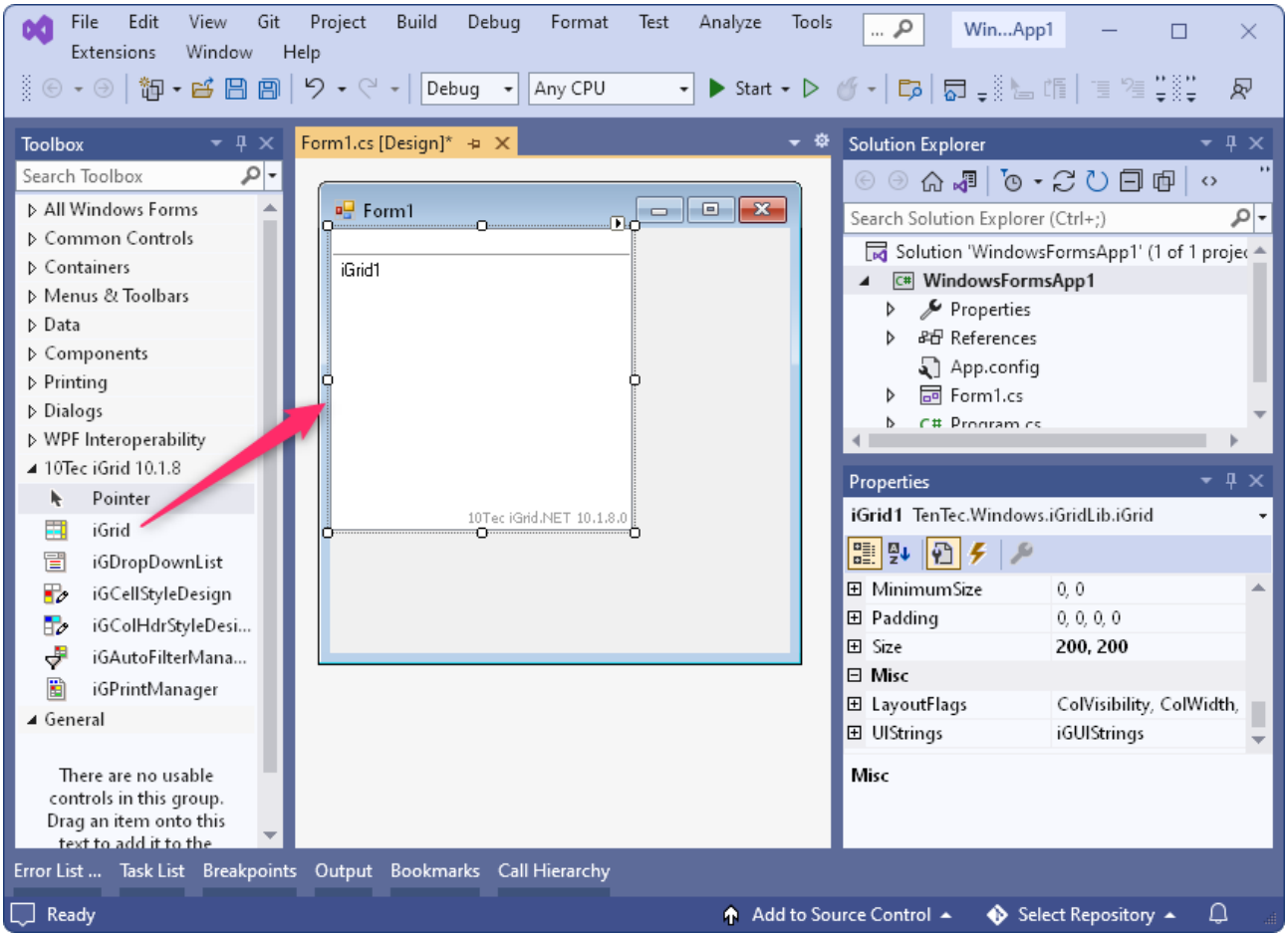
iGrid's approach in managing group rows is based on the concept of row levels: a row's level determines its position in the grouping hierarchy. Row levels are set automatically by iGrid when you group its rows, but you can also create your own row hierarchy manually. Row levels are also used in rows with normal cells to create tree-view grids with automatic support for sorting without a lot of coding!

iGrid was designed and optimized to work without data sources and provides the best tools for building non-data-bound interfaces. But this does not mean you cannot view and edit data from data sources. iGrid allows you to upload data from ADO.NET data source with one method call. After that, you can easily browse, format, sort, group, and filter database data. If you need full data binding with editing support, iGrid's rich object and event model allows you to implement any data driven interface you exactly need without being limited by a hard-coded data-binding approach.

1.2. How to Start

iGrid.NET is a visual control for the WinForms package in .NET. As a rule, iGrid instances are created at design time by dragging the iGrid component item from the Visual Studio Toolbox onto a form. To make the component item available on the Toolbox, you have to do some preparations. These and other preparatory steps that help you to use the component are performed automatically by the demo setup package, so that it is strongly recommended to install the demo of iGrid.NET before its usage.

If you are developing an application for the classic .NET Framework of the versions 4.0-4.8, the main iGrid component item and other items from the iGrid library automatically appear on the Toolbox after installing the demo. You will see them in a new Toolbox tab after installing the demo of the corresponding version. All what you need to create an instance of iGrid on a form is to double-click its item in the Toolbox tab — an iGrid instance will be create on the form by the Windows Designer:



To show the component items on the Toolbox in projects targeting .NET 6 or a later version of .NET, you must add the corresponding NuGet packages to your project first. They are also installed by the demo installer. If you did not change the default installation settings, you will find them in the following directory:

C:\10Tec\NuGetPackages

A step-by-step guide of adding iGrid NuGet packages to your project and other non-standard situations related to the usage of iGrid component items on the Visual Studio Toolbox are described in greater detail in the Toolbox Items Management Guide deployed by the demo installer.

1.3. Assemblies, Namespaces, and Code Samples

Namespaces in .NET are used to organize types provided by .NET assemblies. iGrid.NET, being a .NET assembly, also uses this concept for its types. There are 3 main namespaces you will use while writing code for iGrid:

- **TenTec.Windows.iGridLib**. This is the main namespace containing the types provided by the core grid control.
- **TenTec.Windows.iGridLib.Filtering**. The types from this namespace are used when you enable the filtering functionality by adding the AutoFilterManager add-on to your project.
- **TenTec.Windows.iGridLib.Printing**. The types from this namespace are used when you enable the printing functionality by adding the PrintManager add-on to your project.

These namespaces are implemented in the 3 assemblies whose file names start with the corresponding namespace. For example, the core grid control functionality of the version 14.0.0 compiled for .NET Framework 4 is contained in the assembly TenTec.Windows.iGridLib.14.0.0.net4.dll. The filtering functionality corresponding to the same version of iGrid is in the assembly TenTec.Windows.iGridLib.Filtering.14.0.0.net4.dll, and so on.

When you create an instance of iGrid on your form by dragging the component icon from the Visual Studio Toolbox onto the form surface, the reference to the corresponding assembly is automatically added to your project by Visual Studio. However, you still need to fully qualify types from the iGrid assemblies in your code. Below is an example of a code snippet in which we get a reference to an **iGCol** object representing the third column and set the word-wrap formatting option for its cells:

```
TenTec.Windows.iGridLib.iGCol myCol = iGrid1.Cols[2];
myCol.CellStyle.TextFormatFlags =
  TenTec.Windows.iGridLib.iGStringFormatFlags.WordWrap;
```

Inserting required namespaces every time makes coding tedious and leads to worse code readability. To improve this, we recommend that you import the used iGrid namespace by adding the corresponding statement at the top of code modules, for example:

```
using TenTec.Windows.iGridLib;
```

All samples in this documentation imply that the corresponding iGrid namespace is already imported. Knowing this, the previous code snippet can be rewritten in a shorter form:

```
iGCol myCol = iGrid1.Cols[2];
myCol.CellStyle.TextFormatFlags = iGStringFormatFlags.WordWrap;
```

The code samples in this documentation are provided mainly in one programming language, C#, not to bloat topics a lot. We tried to use simplest constructions so that the code samples can be easily converted to other programming languages manually or with online code converters. Some conversion tips for Visual Basic, which is the second popular language in the .NET world, are below.

To import iGrid namespaces into your code module, use the VB **Imports** statement at the top of the module:

```
Imports TenTec.Windows.iGridLib
```

When you access indexed collections, square braces are replaced with parentheses. Semicolon as statement separator is not used in VB, so simply remove it. The previous C# code snippet will look like the following in VB:

```
Dim myCol As iGCol = iGrid1.Cols(2)
myCol.CellStyle.TextFormatFlags = iGStringFormatFlags.WordWrap
```

Pay attention to the assignment statement. You use the **Dim** statement in VB for this. Creation of a new object from the iGrid library also looks simpler in VB because the type name is used once in the assignment statement. Compare the following C# and VB statements to create a new iGrid cell style object:

```
iGCellStyle myCellStyle = new iGCellStyle();
Dim myCellStyle As New iGCellStyle()
```

The conversion of control structures like **if** or **for** is also trivial and does not contain any pitfalls.

2. IGRID ESSENTIALS

2.1. Populating iGrid

To populate iGrid.NET, first create the required columns and rows. The cells for them will be created automatically, and they will be ready to store and display values.

iGrid.NET can be populated both at design time and at run time. At design time, you simply use the visual column and row collection editors. At run time, you populate iGrid by creating and accessing its columns, rows and cells from code. The reminder of this topic describes the basic principles of populating iGrid at run time.

In most cases the **Cols**, **Rows**, and **Cells** properties of iGrid are used to populate it. These properties return the collections that represent the columns, rows, and cells respectively.

The main way of creating columns is calling one of the overloaded versions of the **Add** method of the column collection returned by the **Cols** property. The variations of the **Add** method allow you to specify different column parameters, such as header text and width. You can also create several columns at once by changing the **Count** property of the **Cols** collection.

To create rows in iGrid, you should use either the **Add** method or the **Count** property of the **Rows** collection property. They work similarly to the members of the column collection object described above.

After the columns and rows are created, you can populate cells using the **Cells** collection. The items of this collection are accessed by numeric indexes of rows and columns. Indexing of items in all iGrid collections starts from zero — exactly as it is done in .NET. Thus, the index of the last item of any collection will be equal to the number of items in the collection minus 1.

Every iGrid cell is represented with an instance of the **iGCell** object. This object is returned when you access iGrid cells through its **Cells** collection. The properties of the **iGCell** object are used to set properties of the corresponding cell. For example, the background color of the third cell in the second row can be set like this:

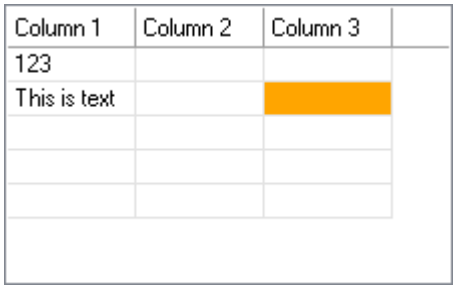
```
iGrid1.Cells[1, 2].BackColor = Color.Orange;
```

The value displayed in a cell is specified with the **Value** property of the cell. This property is of the **Object** type, which means it can store a value of any data type.

The following example creates an iGrid with three columns and five rows. It assigns an integer and a string to cells in the first column and changes the background color of a cell in the third column to orange:

```
iGrid1.Cols.Add("Column 1");
iGrid1.Cols.Add("Column 2");
iGrid1.Cols.Add("Column 3");

iGrid1.Rows.Count = 5;

iGrid1.Cells[0, 0].Value = 123;
iGrid1.Cells[1, 0].Value = "This is text";
iGrid1.Cells[1, 2].BackColor = Color.Orange;
```

The screenshot below shows the resulting grid:



2.2. String Keys for Columns and Rows

One of the best features of iGrid that can simplify coding and greatly speed up your code performance is the ability to assign string keys to columns and rows and use them to access these objects. This works as if you gave a name to a column or row and then quickly find it by its name. String keys give you the following benefits:

- If you access a column (row) by its string name, your code becomes much more self-descriptive and understandable.
- When you use a numeric index to access a column (row), you will need to change all statements referencing this column (row) in the case if you add or remove a column (row) before the referenced one. In the case of a string key your code will remain the same.
- iGrid provides a super-fast access to rows when you access them by string keys due to the internal index of keys. This allows you to quickly access any row in the grid by its string name after operations that reorder grid rows, such as sorting, grouping, etc. This works extremely fast even in grids with 100'000+ rows.

Column and row keys can be set when you create a column or row using the corresponding overloaded versions of the **Add** methods of the **Cols** or **Rows** collections of iGrid. For example, the following statement creates a column with the key "id" and title "Customer ID" the user will see on the screen:

```
iGrid1.Cols.Add("id", "Customer ID");
```

Keys can also be set, changed or retrieved at any time using the **Key** property of the **iGCol** and **iGRow** classes representing columns and rows. Instances of these classes can be retrieved from the **Cols** and **Rows** collections of iGrid. For example, the statement below sets the key of the 3rd row of iGrid to "pos":

```
iGrid1.Rows[2].Key = "pos";
```

Having column or rows keys defined, we can use them in many overloaded versions of iGrid members including the **Cells** collection of iGrid. For instance, if the 4th column in a grid iGrid1 had the key "id", we could change the value of the 3rd cell in this column using the statement

```
iGrid1.Cells[2, 3].Value = "Q1234";
```

or an equivalent statement with the column string key:

```
iGrid1.Cells[2, "id"].Value = "Q1234";
```

The same is applicable to rows with string keys:

```
iGrid1.Cells["pos", 3].Value = "Q1234";
```

Or we can even use both row key and column key accessing the required cell:

```
iGrid1.Cells["pos", "id"].Value = "Q1234";
```

Column and row keys are case-insensitive in iGrid.

2.3. Screen Updates and Performance

By default, iGrid redraws its contents after each change to its cells. If you make a large number of such changes, this can significantly reduce performance. To achieve the best performance, it is strongly recommended to wrap the code that modifies cells between calls to the **BeginUpdate** and **EndUpdate** methods. **BeginUpdate** disables redrawing after each change, which can greatly improve performance. After all changes are complete, call **EndUpdate** to resume updating and redraw the grid, so that all recent changes are displayed in a single paint cycle.

The following example populates cells with their row and column indexes using the template "R<row_index>C<column_index>":

```
foreach (iGCell cell in iGrid1.Cells)
{
  cell.Value = $"R{cell.RowIndex}C{cell.ColIndex}";
}
```

The following version of this code performs much better for the reason described above:

```
iGrid1.BeginUpdate();

foreach (iGCell cell in iGrid1.Cells)
{
  cell.Value = $"R{cell.RowIndex}C{cell.ColIndex}";
}

iGrid1.EndUpdate();
```

Calls to **BeginUpdate** and **EndUpdate** are cumulative. If **BeginUpdate** is called multiple times, updating is resumed only after the matching final call to **EndUpdate**.

2.4. Cell Style Objects

Understanding cell styles

One of the key features of iGrid is the ability to use cell styles to apply the same formatting to several cells. This works like text styles in Microsoft Word.

Consider the following code snippet in which we set the background color and text alignment for the first two cells in the first column:

```
iGrid1.Cells[0, 0].BackColor = Color.Coral;
iGrid1.Cells[0, 0].TextAlign = iGContentAlignment.TopRight;

iGrid1.Cells[1, 0].BackColor = Color.Coral;
iGrid1.Cells[1, 0].TextAlign = iGContentAlignment.TopRight;
```

This work can be done with a cell style object. An iGrid cell style object is an instance of the **iGCellStyle** class. You can create it in code or at design time, and then apply to the required cells using the **Style** property of the cells.

We can create a cell style object to format the cells from the code snippet above:

```
iGCellStyle myStyle = new iGCellStyle();
myStyle.BackColor = Color.Coral;
myStyle.TextAlign = iGContentAlignment.TopRight;
```

Having this style object, we can apply it to our cells:

```
iGrid1.Cells[0, 0].Style = myStyle;
iGrid1.Cells[1, 0].Style = myStyle;
```

The two main benefits of this approach is that your code becomes simpler and your formatting settings are done only in one place. For example, if later you decide to change the background color of our cells to lime, you will need just one statement:

```
myStyle.BackColor = Color.Lime;
```

To increase performance, iGrid does not redraw all its cells linked to a style after you change the style properties. This occurs only during the next redrawing of iGrid. If you want to reflect your changes in the style object immediately, request the drawing of the grid control with the **Invalidate** method:

```
iGrid1.Invalidate();
```

Formatting with styles is extremely useful if you use the same formatting for a huge number of cells. The operation of changing formatting for all those cells takes almost no time and does not depend on the number of involved cells.

Storing cell formatting in a style object

Every iGrid cell implements various formatting properties like **ForeColor**, **Font**, **TextAlign**, etc. All these properties are mapped to the internal cell style object used to format a cell. If you change one of these properties, actually you change the corresponding property of the internal style object attached to the cell.

By default a cell does not have this internal style object. It is created automatically during the first assignment to a formatting property of the cell. You can access this cell style object using the **Style** property of the cell.

This approach with storing custom cell formatting in a style object allows us to save a lot of RAM because non-formatted cells use no memory to store formatting settings. The memory is used only to store the data-related cell properties like **Value**.

If several cells should use the same formatting, it is advisable to create a cell style object with the required formatting settings explicitly and apply it to those cells using their **Style** property as we do it in the beginning of this topic. If you go another way and assign the same values to the same formatting properties of those cells, iGrid will create internal style objects for those cells automatically. They will be identical, but every cell will contain a separate instance of such an object. Obviously, we will waste RAM in this case and our code will work slower compared to the approach based on a cell style.

Formatting cells of a whole column

As a rule, in real-world applications we use the same formatting for all cells belonging to one column. iGrid allows you to do that very easily as every column already has a cell style object used to format its cells if they do not have own styles. This is the column's **CellStyle** object, which exists for every column in iGrid. For example, you can apply the formatting like the above to all cells in the first column using the following code:

```
iGrid1.Cols[0].CellStyle.BackColor = Color.Coral;
iGrid1.Cols[0].CellStyle.TextAlign = iGContentAlignment.TopRight;
```

Note that in this case cells in the column use no memory to store their formatting settings. they simply inherit the formatting from the column's cell style object.

If several columns should use the same formatting, you can save even more RAM and simplify your code if you create the corresponding **iGCellStyle** object explicitly and assign it to the **CellStyle** property of the required columns:

```
iGrid1.Cols[0].CellStyle = myStyle;
iGrid1.Cols[1].CellStyle = myStyle;
```

All these principles related to style objects for normal cells are applicable to footer cells and column headers as well. You can find out more about cell style inheritance in iGrid from the Cell Style Inheritance topic.

2.5. Cells Selection Basics

In iGrid you can have the current cell and selected cell(s), and in the general case it is not the same. The current cell is the cell to which the keyboard input is directed, whereas selected cells are the cells marked with the special selection color. We can have only one current cell in iGrid, but there can be a lot of selected cells.

iGrid with the default settings has only one selected cell which is the current cell. You can change this behavior and enable a multi-selection mode with the **SelectionMode** property.

iGrid provides a special "row mode" in which the entire row is selected (or entire rows when multiple selection is enabled). Row mode can be activated with the **RowMode** property. Note that when row mode is activated, the current cell is not displayed, and the current row as a whole behaves as the current cell for navigation purposes. This behavior is controlled by the **RowModeHasCurCell** property, which is False by default. Set this property to True if you want the current cell to remain visible in row mode.

iGrid stores two separate selection object collection — selected cells and selected rows. The selected cells can be enumerated using the **SelectedCells** property; the selected rows can be enumerated with the **SelectedRows** property. For more information about selection, read Chapter Selection and the Current Cell.

2.6. Browse Mode and Edit Mode

Browse mode

When a form containing iGrid appears on the screen, iGrid starts in browse mode. This means that all keyboard commands will be treated as commands that control the current cell (or the current row if row mode is on). For instance, the RIGHT ARROW key will move the current cell right. The table below lists all main keyboard commands available in browse mode:

KEY	CELL SELECTION MODE	ROW SELECTION MODE
UP ARROW	Move the current cell up.	Move the current row up.
DOWN ARROW	Move the current cell down.	Move the current row down.

KEY	CELL SELECTION MODE	ROW SELECTION MODE
LEFT ARROW	Move the current cell left.	Scrolls the grid horizontally to the left if it has the horizontal scroll bar.
RIGHT ARROW	Move the current cell right.	Scroll the grid horizontally to the right if it has the horizontal scroll bar.
HOME	Move the current cell to the first column.	Move the current row to the first row.
END	Move the current cell to the last column.	Move the current row to the last row.
CTRL+HOME	Move the current cell to the first column in the first row.	Move the current row to the first row.
CTRL+END	Move the current cell to the last column in the last row.	Move the current row to the last row.
PAGE UP	Scroll the grid one page up and move the current cell respectively.	Scroll the grid one page up and move the current row respectively.
PAGE DOWN	Scroll the grid one page down and move the current cell respectively.	Scroll the grid one page down and move the current row respectively.

If multi-selection mode is on, you can use additional modifier keys like CTRL and SHIFT to select/deselect several cells or rows. The detailed information about these features can be found in the Selection Modes in iGrid topic.

Edit mode

To put iGrid in edit mode, start editing of the current cell by pressing ENTER or F2. For text cells, you can also press a key that enters a letter, digit, symbol, punctuation mark, or the space character. For combo box cells, you can use the F4 key that opens the cell drop-down list. iGrid returns to browse mode after you have committed or cancelled changes made during editing. All the commands available in edit mode can be found in the Keyboard and Mouse Editing Commands topic.

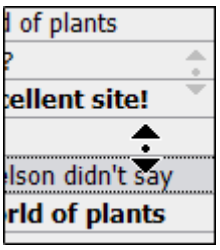
2.7. Scrolling iGrid Contents

When you change the current cell or row using keyboard commands, the new current cell or row can be outside of the viewport. In this case iGrid will scroll its contents automatically trying to fully display the new current cell or row in the viewport.

You can scroll iGrid without changing the current cell or row using different ways. First, you can do this by operating the scroll bars with your mouse. Second, you can scroll the iGrid contents using the mouse wheel. If you rotate the mouse wheel without pressing a modifier key, the contents will be scrolled in the vertical direction. If you hold down the SHIFT key while rotating the wheel, the contents will be scrolled in the horizontal direction.

iGrid provides you with one more way to scroll its contents called "autoscrolling". To start autoscrolling, (1) push the middle button on your mouse and drag the mouse pointer holding down the button or (2) click the middle mouse button over the grid and move the mouse pointer. In the first case autoscrolling will continue until you release the button. In the second case the action will continue until you click any mouse button, press a key or activate another window. If autoscrolling

is activated, the origin point is marked with a dimmed image and the cursor changes to indicate available scrolling directions:



Note that some mouse drivers allow you to assign custom actions to the middle mouse button. Mouse wheel scrolling and autoscrolling will work only if the middle mouse button performs its default action.

If iGrid is displayed on a tablet or a screen with touch support, you can scroll the iGrid contents with a finger using the traditional swipe gesture.

2.8. Rendering Styles, Color Modes, and Themes

Let's define the following three terms used to describe the appearance of iGrid in this documentation:

- Rendering style — the visual style used to draw iGrid user interface elements, such as system, classic, or flat.
- Color mode — the overall light or dark appearance of the control.
- Theme — the complete visual appearance of the control, determined by the combination of its rendering style and color mode.

iGrid allows you to select one of the three predefined rendering styles — the system style, the classic 3D style, and the flat style. This is done with the help of the **RenderStyle** property of iGrid. It accepts a value from the **iGRenderStyle** enumeration with the three values: **System**, **Classic**, and **Flat**. The system rendering style makes the control look as native to the operating system as possible, whereas the classic 3D and flat styles make the grid's appearance independent of the operating system and provide full control over the colors of its elements. You can find out more about these features from the topics in the Rendering Styles sections.

Starting with .NET 10, WinForms allows you to choose the application's color mode by calling the **Application.SetColorMode()** method before the first form is created. In iGrid, the color mode is selected automatically according to that application-wide setting. A separate color mode cannot be specified for an individual iGrid control, which is consistent with the WinForms model and the behavior of its standard controls. All built-in rendering styles fully support both color modes, while the colors of individual elements can still be customized through iGrid properties.

3. MANAGING COLUMNS

3.1. Creating and Removing Columns

Most methods and properties for manipulating columns are provided by the **Cols** collection of iGrid.

The easiest way to create new columns is to set the **Count** property of the **Cols** collection to the required number of columns, for example:

```
iGrid1.Cols.Count = 5;
```

After that you can access individual columns in the **Cols** collection and change their default parameters if required like in the following code:

```
iGCol myColumn = iGrid1.Cols[0];
myColumn.Text = "Column caption";
myColumn.Width = 100;
```

You can also create new columns one-by-one using the **Add** or **AddRange** methods of the **Cols** collection. Various overloaded versions of these methods allow you to specify column properties in these method calls. For example, the previous code snippet can be rewritten as follows:

```
iGrid1.Cols.Add("Column caption", 100);
```

The **Add** and **AddRange** methods add new columns after the last existing column. To create new columns before an existing one, use the **Insert** or **InsertRange** methods of the **Cols** collection.

When you create a new column, iGrid uses the settings from its **DefaultCol** property to initialize the properties of the new column. Below is an example demonstrating how to set the default width of all new columns to 100 pixels:

```
iGrid1.DefaultCol.Width = 100;
```

The **DefaultCol** object contains several object properties, such as **CellStyle**. They are cloned into new instances of the **iGCol** class representing new columns during their creation with the approaches described above.

Columns can be removed with the **RemoveAt** or **RemoveRange** methods of the **Cols** collection. You can also remove several last columns simply by decreasing the **Count** property of the **Cols** collection.

3.2. Accessing Columns

iGrid columns can be accessed through the **Cols** collection. Each column optionally can have its own string key. The **Cols** collection is indexed by integer values (column index, starts with zero) or by string column keys. Column keys are case-insensitive.

The row text column is accessed through the **RowTextCol** property of the grid. You can also access it through the **Cols** property by passing -1 as the column index.

Access to the properties of a column is provided through the **iGCol** class. An object of this class that represents a single column is just a reference to the actual column data in the internal iGrid array (in fact, it stores the column index). Note that after you add or remove columns, the actual column may change its index and an **iGCol** object will reference an improper column data.

3.3. Column Properties

Each column has a set of properties which define the behavior and appearance of the whole column. This set is listed below:

PROPERTY	DESCRIPTION
AllowGrouping	Determines whether rows of the grid can be grouped by this column. This property acts only when you try to group the grid through its visual interface (with the group box).
AllowMoving	Determines whether the order of the column can be changed. When this property is set to False, you cannot move this column both through visual interface and through code.
AllowSizing	Determines whether the column can be resized through visual interface. Also it acts when the **AutoWidth** method is invoked.
Cells	Provides access to the cells of the column.
CellStyle	Gets or sets the style which determines the appearance and behavior of the cells in the column. If a cell from the column does not have its own style, or some of its properties are not set, and the row containing the cell does not have a style attached to it too or row style properties are not specified, the cell will use the corresponding setting from this column cell style.
ColHdrStyle	Gets or sets the header style object which determines the appearance and behavior of the header of the column. If the column header does not have its own style or some of its properties are not set, it will use the **ColHdrStyle** of the column.
CustomGrouping	Gets or sets a value determining whether to apply custom grouping to the column.
DefaultCellAuxValue	Gets or sets the value that will be set to the **AuxValue** property of the new cells of the column.
DefaultCellImageIndex	Gets or sets the default image index for new cells in the column.
DefaultCellValue	Gets or sets the default cell value for new cells in the column.
GroupIndex	Gets the zero-based index of the column within the group object or -1 if it is not grouped.
GroupSummaryType	Gets or sets the in-column group summary type.
ImageIndex	Gets or sets the image index of the column header. In fact this property provides access to the image index of the column header in the first header row in the multi-row header.

PROPERTY	DESCRIPTION
IncludeInSelect	Gets or sets a value that determines whether the cells of this column can be selected. If a column is not included in selection and you click a cell in this column, the selection is changed as if the mouse pointer was in the cell in this row and in the column where the current cell was in before.
Index	Gets the index of the column. The index of the column can be changed only by adding or removing columns.
IsRowText	Indicates whether this column is the row text column. The row text column is a column which is used for drawing the group rows and the text beneath the normal cells (like the message preview in MS Outlook).
Key	Gets or sets the string key of the column. You can easily access the column by its string key instead of its numeric index. String keys are stored in a sorted array and provide fast access to a column.
MaxWidth	Gets or sets the maximum width of the column. By default this value is -1. It means that the maximum width is not limited.
MinWidth	Gets or sets the minimum width of the column. By default this value is -1. It means that the minimum width is not specified and is limited by zero.
Order	Gets or sets the visual order of the column. This order can be changed both through visual interface and code.
Pattern	Allows to set and get the majority of the column properties at once. To set a column's properties, define an **iGColPattern** object and assign it to this property.
SortIndex	Gets the sort index of the column. By default the sort index of the column is set to -1. It means that the grid is not sorted by this column. Sort index can be changed both through visual interface and through code.
SortOrder	Gets or sets the default sort order of the column (ascending/descending). This sort order is used by iGrid to sort a non-sorted column when the user clicks its header.
SortType	Gets or sets the sort type of the column. The sort type of the column determines the rule (by cell values, by cell icons, etc.) used to sort the column.
Tag	Provides an extra property which can be used on your own.
Text	Gets or sets the text displayed in the header of the column. In fact this property provides access to the text of the column header in the first header row in the multi-row header.
Visible	Gets or sets a value indicating whether the column is visible.

PROPERTY	DESCRIPTION
Width	Gets or sets the width of the column. The width of the column is limited by **MinWidth** and **MaxWidth**. You cannot set the width of the column less than min width or greater than max width.
X	Gets the x-coordinate of the left edge of the column in the client coordinates of the grid.

3.4. Adjusting Column Headers

Every iGrid column is represented with an **iGCol** object you can retrieve using different methods (see the Accessing Columns topic). The **iGCol** properties determine the general column look (width, text, sort index) and behavior (the ability to resize, move, sort, etc.). If you need to change the text displayed in a column's header, you can do that using the **iGCol**.**Text** property. The following code sets a new column header text for the first column:

```
iGrid1.Cols[0].Text = "New column header text";
```

To access other properties which define column header look — color, font, etc. — you need to use the collection of header cells. iGrid can have several rows in the header too, and you access the header cells using the two-dimensional **Header**.**Cells** array. When you access a header cell, the first index is the row and the second index is the column. In most cases iGrid has one header row, so the row index equals zero.

Here is a code snippet to change the header text color and its text for the first column:

```
iGrid1.Header.Cells[0, 0].Value = "Another text";
iGrid1.Header.Cells[0, 0].ForeColor = Color.Red;
```

As one more approach, you can change the style of all column headers within one column using the **iGCol**.**ColHdrStyle** property. For example, to set the font for the first column header to bold, use code like this:

```
iGrid1.Cols[0].ColHdrStyle.Font = new Font(iGrid1.Font, FontStyle.Bold);
```

iGrid's column headers have a lot of features. They are described in detail in Chapter Header Section and Column Headers.

3.5. Column Pattern

All the properties of a column can be obtained with its **Pattern** property at once. The value returned by this property is of the **iGColPattern** type. An instance of this class just stores column properties but these properties are not linked to any real column; if you change them, nothing happens. The column pattern may be useful if you want to copy the properties of a column to another column or when you want to create a group of similar columns with most of the properties identical.

For example if you want to create 5 columns with identical parameters except for the caption, you can do the following:

```
//Create a pattern.
iGColPattern myPattern = new iGColPattern();

//Set properties common to all the columns.
myPattern.Width = 100;
myPattern.CellStyle.BackColor = Color.Green;
...

//Add the columns using the pattern.
iGrid1.Cols.Add("Col1", myPattern);
iGrid1.Cols.Add("Col2", myPattern);
iGrid1.Cols.Add("Col3", myPattern);
iGrid1.Cols.Add("Col4", myPattern);
iGrid1.Cols.Add("Col5", myPattern);

//The above version of the Add method takes
//all the properties except the column name
//from the pattern. The call
//iGrid1.Cols.Add("Col1", myPattern);
//is equivalent to the following:
//myPattern.Text = "Col1";
//iGrid1.Cols.Add(myPattern);
```

Also you can copy all the properties of one column to another:

```
iGrid1.Cols[3].Pattern = iGrid1.Cols[0].Pattern;
```

Notice that a new instance of the **iGColPattern** class that represents a column is created each time when you get the value of the **Pattern** property of the column (in fact, iGrid does not store an **iGColPattern** object for each column — it is created on fly). When you set the value of this property by assigning an instance of the **iGColPattern** class, again, the reference to the specified **iGColPattern** object isn't saved but the properties of this object are used to change the corresponding column properties. When you change properties of an object returned by the **Pattern** property, nothing changes in the column.

3.6. Changing Column Order

To change column order from code, use either the **Move** method or the **Order** property of the **iGCol** object. The **Order** property allows you to move one column at once whereas the **Move** method allows you to move several columns at one go.

The order of a column cannot be changed if its **AllowMoving** property is set to False. Also, you cannot move a column if its movement will involve changing order of a non-movable column. For example, if the first column is non-movable, and you try to move the second column to the first position in code, the gird will raise an exception.

Furthermore, there are limitations on changing the order of columns with headers merged horizontally. Columns with merged headers can only be moved within the merged header. But you can change the position of all the columns with merged header as a single column group. For example if the headers of the 2nd and 3rd columns are merged, you can move the 2nd and 3rd columns together. One more limitation related to merged column headers is that merged column headers cannot be separated by the frozen columns edge.

To determine whether a column or a group of adjacent columns can be moved, use the **CanMove** method of the **iGCol** object. To determine whether a column or columns can be placed to a certain position, use the **CanPlaceTo** method.

For example, if you want to know whether the first and second columns together can be moved to the fifth position, you can use one of the following approaches.

Example 1:

```
// Column movement limits in the iGCol.CanPlaceTo() method
int myStartOrder, myEndOrder;

// Map used in the iGCol.CanPlaceTo() method
bool[] myRowsMap;

// Get the first column
iGCol myCol = iGrid1.Cols.FromOrder(0);

myCol.CanMove(2, // number of columns to move
  out myStartOrder, out myEndOrder, out myRowsMap);
if (myStartOrder <= 4 && myEndOrder >= 4)
{
  if (myCol.CanPlaceTo(
    4, // new column order
    2, // number of columns to move
    myRowsMap))
  {
     // The columns can be moved to the fifth position
     ...
  }
}
```

Example 2:

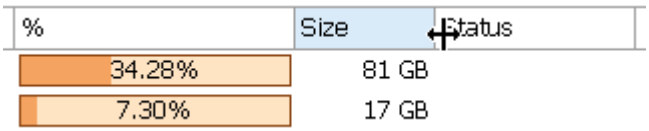
```
// Get the first column
iGCol myCol = iGrid1.Cols.FromOrder(0);

if (myCol.CanPlaceTo(4, 2))
{
  // The columns can be moved to the fifth position
  ...
}
```

The examples above are equivalent.

iGrid allows you to move columns through its visual interface by dragging column headers. To prohibit this operation or to limit the possible places a column can be dragged to, use the **ColHdrStartDrag**, **ColHdrDragging** and **ColHdrEndDrag** events. The **ColOrderChanged** event is raised after the column order has been physically changed.

3.7. Resizing Columns

The user can change the width of a column interactively by dragging the corresponding column header divider. This is done in the special area around the right edge of the column header. When the mouse pointer is in this area, it changes to a two-directional arrow indicating the ability to resize the column:



You can limit the range of acceptable column sizes using the **MinWidth** and **MaxWidth** property of the **iGCol** object representing the column. The Boolean **AllowSizing** property of this object can be used to disable the resizing of the column.

To change the width of a column from code, assign the new width in pixels to the **Width** property of the corresponding **iGCol** column object.

iGrid raises the **ColWidthStartChange**, **ColWidthChanging**, and **ColWidthEndChange** events before, during, and after the user changes a column width. These events provide you with the row and column indexes of the column header used for resizing. iGrid supports resizing of a group of columns at once if these columns have a merged header and you drag its divider.

The distance from the column divider in which column resizing is possible can be adjusted with the **ColResizeAreaExtentToDivider** property of iGrid. The related property, **ColResizeAreaProtectedSpace**, can be used to specify the width of the part of the column in which resizing is not possible — to provide the ability to click the column header if it is narrow enough.

The **ImmediateColResizing** property of iGrid determines whether iGrid should update the column contents after every mouse move event while the user is resizing this column. The default value of this property is True, which may be undesirable if the cells contain heavy-to-draw content. To turn repainting off during interactive column resizing, set **ImmediateColResizing** to False.

3.8. Automatic Adjustment of Column Width

iGrid allows you to adjust the width of a column automatically so that the contents of its cells will be displayed without clipping. This can be done interactively and from code.

To automatically adjust the width of a column interactively, double-click the divider of the column's header. You can prohibit this behavior for individual columns with the **ColDividerDoubleClick** event of iGrid: set its **DoDefault** property to False to prevent the default action from taking place.

To automatically adjust the width of a column from code, call the **AutoWidth** method of the corresponding column object (**iGCol**).

Pay attention to the fact that any instance of iGrid has the so-called row text column used to store the contents of row text cells and group rows (see the Row Texts section for more information). You can access the row text column with the **RowTextCol** property of iGrid. It returns an instance of the **iGCol** class, and thus you can also call the **AutoWidth** method for the row text column like for a normal column. In this case iGrid will adjust the widths of the normal columns in which row text cells are displayed. Note that the widths of the normal columns can decrease during this process. If this effect is undesirable, you can use the overloaded version of the **AutoWidth** method with the **allowDecrease** parameter to disallow column width reduction. Another overloaded version of the **AutoWidth** method with the **rowTextKinds** parameter can be used to specify which row text kinds, row text cells and/or group rows, should be processed.

The **iGrid**.**Cols** property returns the column collection object that also implements its own **AutoWidth** method. When you call it, iGrid automatically adjusts the width of all columns. If iGrid displays row texts inside rows, the width of the columns containing row text cells will be adjusted to display both the contents of the normal cells and row text cells.

Cell text wrapping is turned off by default, and iGrid lines up cell text in one line to calculate the optimal column width. If you enable word wrapping for cell texts by specifying the **WordWrap** flag in the **TextFormatFlags** property of the cells, iGrid will take into account the row height while calculating the optimal width for every cell. It will determine how many integral text lines can be drawn in the cell and calculate the minimum width to display the cell text without clipping in this number of integral text lines.

The aforementioned **AutoWidth** methods also take into account the CR/LF characters you can use to forcibly create text lines in cell texts. If the number of the text lines created with CR/LF is less

than the number of text lines that can be placed in the grid row a cell belongs to and the cell uses word wrapping, iGrid tries to minimize the width of the cell by wrapping the text lines; otherwise each text line is lined up in one line to display the maximum number of text lines regardless of the word wrapping setting for the cell.

Some notes about automatic column width adjustment:

- When the width of a column is adjusted automatically, by default iGrid takes into account the contents of column's header to display its contents without clipping too. You can change this behavior with the **AutoWidthColMode** property that accepts one of the following values: **Cells**, **Header**, and **HeaderAndCells**.
- You can limit the range of acceptable sizes of a column using the **MinWidth** and **MaxWidth** property of the **iGCol** object representing the column.
- iGrid supports the "auto-width" operation for custom-drawn cells too. To determine the optimal width for such cells, iGrid raises the **CustomDrawCellGetWidth** event. You should provide the width of your custom cell contents in a handler of this event.

If iGrid contains merged cells, you can control how they are processed during automatic column width adjustment with the help of the **AutoWidthColSpanHandling** property of iGrid. This capability is described in greater detail in the Merged Cells in Auto-sizing Operations topic in the section devoted to merged cells.

When iGrid automatically adjusts the width of a column, it decides whether to process a cell of the column based on the visibility of the corresponding row. The **AutoWidthColCheckRowVisibility** property of iGrid allows you to control how iGrid checks row visibility during automatic column width adjustment. This property accepts a combination of flags from the **iGRowVisibilityProperties** enumeration. These flags — **Visible**, **VisibleFiltered** and **VisibleParentExpanded** — correspond to the three row properties that determine row visibility. If a flag is set, iGrid will check the value of the corresponding row property while adjusting the column width. In most cases a row should be skipped if it is currently not visible, except the case when the row belongs to a collapsed group row or tree node. The default value of the **AutoWidthColCheckRowVisibility** property is the combination of the **Visible** and **VisibleFiltered** flags, which corresponds to the described typical behavior.

3.9. Resizing All Columns to Fill Grid Width

iGrid offers a mode that automatically resizes all columns to fill the entire grid width — when there is no extra space after the last column and the horizontal scroll bar is not needed. When the grid size changes, the width of every column is increased or decreased proportionally to fit the available space. This mode is activated by setting the **AutoResizeCols** property to True.

iGrid takes into account the values of the following column properties when it automatically recalculates column widths in this mode. If the **AllowSizing** property is set to False for a column, its width remains unchanged. If a column can be resized, but there are restrictions on its minimal and maximal width specified in the **MinWidth** and **MaxWidth** properties, they are taken into account. And iGrid resizes only the visible columns in column auto resize mode.

If you do not want to enable this mode but want to automatically resize columns once from code to occupy the whole available client width (for example, after initial population of the grid), call the **DoAutoResizeCols** method.

4. MANAGING ROWS

4.1. Creating and Removing Rows

Most methods and properties for manipulating rows are provided by the **Rows** collection of iGrid.

The easiest way to create new rows is to set the **Count** property of the **Rows** collection to the required number of rows. For example, the following code creates 5 rows in an empty iGrid:

```
iGrid1.Rows.Count = 5;
```

You can also create new rows with the **Add** or **AddRange** methods of the **Rows** collection, for example:

```
iGrid1.Rows.Add();
```

The **Add** method returns an instance of the **iGRow** class representing the added row. You can use it to set the properties of the new row. For example, you can set the height of the new row to 30 pixels this way:

```
iGRow myRow = iGrid1.Rows.Add();
myRow.Height = 30;
```

An **iGRow** object returned by the **Add** method can also be used to set values for cells in this row:

```
iGRow myRow = iGrid1.Rows.Add();
myRow.Cells[0].Value = "Text"; // value of cell in column 0
myRow.Cells[1].Value = 100;    // value of cell in column 1
myRow.Cells[2].Value = 200;    // value of cell in column 2
```

The iGrid control itself implements the **AddRow** method that can be used to add an empty row and optionally populate its cells with some values in one call. This method can help to make your code shorter and faster. The code sample above can be rewritten as follows using this method:

```
iGrid1.AddRow("Text", 100, 200);
```

When you create new rows using the approaches described above, iGrid uses the values of its **DefaultRow** object property to initialize the corresponding properties of the new row. For example, you can specify the default height of 30 pixels for all new rows this way:

```
iGrid1.DefaultRow.Height = 30;
```

There is an alternative way to specify the properties of a new row — with the **iGRowPattern** class. Instances of this class can be passed to the **Add** and **AddRange** methods of the **Rows** collection. For example, the following code creates a new row with the height of 30 pixels using the row pattern object:

```
iGRowPattern myRowPattern = new iGRowPattern(30);
iGrid1.Rows.Add(myRowPattern);
```

The **Add** and **AddRange** methods add new rows after the last row. To create a new row (rows) before an existing one, use the **Insert** or **InsertRange** method of the **Rows** collection.

Rows can be removed with the **RemoveAt** or **RemoveRange** methods of the **Rows** collection. You can also remove several last rows in the grid simply by decreasing the **Count** property of the **Rows** collection.

4.2. Accessing Rows

iGrid rows can be accessed through the collection returned by the **Rows** property. Each row can optionally have its own string key you can use to find the named row quickly. The row collection is indexed with integer values (row index, starts with zero) or with string row keys. Row keys are case-insensitive.

Access to the row properties is provided through the **iGRow** object class. An object of this class does not contain the row data inside, though this data is set and retrieved with the properties of the **iGRow** object. The **iGRow** object is a lightweight wrapper that stores only the row index and uses this row index to find the corresponding row data in the internal structures of iGrid.

This approach allows iGrid to work fast because it does not create unneeded row objects upfront, which is useless memory consumption if you never access row objects in your code. Row objects are created on the fly only when you access them. However, there is one downside of this approach. Remember that every **iGRow** object you requested remains valid until you insert, remove or reorder rows in iGrid. After doing this, the requested row may change its index and the **iGRow** object will reference the data for another row or even invalid data.

4.3. Row Properties

Each row has a set of properties which define the behavior and appearance of the whole row. This set is listed below:

PROPERTY	DESCRIPTION
Cells	Returns the collection of the cells of the row. Allows you to access cells of the row by column index or column string key.
CellStyle	Gets or sets the style which determines the appearance and behavior of the cells in the row. If a cell from the row does not have its own style, or some of its properties are not set, the cell will use the corresponding setting from this row cell style.
Expanded	Gets or sets value indicating whether the row is expanded or collapsed. Use this property in conjunction with the **Level** property to create tree-like grid.
GroupInfo	Contains main information about the group for which the row was created (group value, group value text, item count, summary values).
HdrBounds	The bounds of the row's header.
HdrGlyph	Indicates which glyph is currently displayed in the row.
Height	Gets or sets the height of the row. The height of the row includes the height of its normal cells and the height of the row text cell.
Index	Gets the zero-based index of the row.
Key	Gets or sets the string key of the row. Using the string key you can easily access a row by its string representation. Keys are stored in sorted array and provide fast access to rows.

PROPERTY	DESCRIPTION
Level	Gets or sets the hierarchy level (0, 1, 2, and so forth) of the row. Generally these values are used to create tree-like grids; these values are also set by iGrid automatically when you group its rows. By default, all rows have the zero hierarchy level.
NormalCellHeight	Gets or sets the height of the cells above the row text cell. This height is included in the total height of the row (its **Height** property).
Parent	Returns the parent group row or parent tree node for the row.
Pattern	Allows to set and get majority of the row properties at once.
RowTextCell	Provides access to the row text cell.
Selectable	Determines whether the row can be selected.
Selected	Determines whether the row is selected.
Sortable	Gets or sets a value indicating whether the row can change its position while sorting.
Tag	Provides an extra property which can be used on your own.
TreeButton	Determines whether a tree button (the plus/minus button) which you can use to collapse/expand the row is visible in the first cell of the row.
Type	Gets or sets the type of the row (normal cell row or a kind of group row).
Visible	Determines whether the row is visible in general. Two other service properties related to row visibility, **VisibleFiltered** and **VisibleParentExpanded**, also affect the row visibility state.
VisibleFiltered	Determines whether the row is visible after applying a filter. This is a kind of a service property that is used by the built-in search-as-type functionality and the AutoFilterManager add-on.
VisibleParentExpanded	Indicates whether the row is visible or hidden when one of its parent rows is collapsed or expanded. This property returns False if the row belongs to a group or tree node that is currently collapsed.
Y	Gets the y-coordinate of the row in the client coordinates of the grid.

4.4. Row Pattern

All the properties of a row can be obtained with the **Pattern** property at once. The value returned by this property is of the **iGRowPattern** type. An instance of this class just stores row properties but these properties are not linked to any real row; if you change them, nothing happens. The row

pattern may be useful if you want to copy the properties of a row to other rows or when you want to create a group of similar rows with most of the properties identical.

For example if you want to create 5 rows with identical parameters, you can do the following:

```
//Create a pattern.
iGRowPattern myRowPattern = new iGRowPattern();

//Set properties common to all the rows.
myRowPattern.Height = 20;
myRowPattern.Type = iGRowType.ManualGroupRow;
myRowPattern.Level = 1;
...

//Add the rows using the pattern
iGrid1.Rows.Add(myRowPattern);
iGrid1.Rows.Add(myRowPattern);
iGrid1.Rows.Add(myRowPattern);
iGrid1.Rows.Add(myRowPattern);
iGrid1.Rows.Add(myRowPattern);
```

Also you can copy all the properties of one row to another with this statement:

```
iGrid1.Rows[3].Pattern = iGrid1.Rows[0].Pattern;
```

Notice that a new instance of the **iGRowPattern** class that represents a row is created every time when you get the value of the **Pattern** property of the row (in fact, iGrid does not store **iGRowPattern** objects for each row — they are created on fly). When you set the value of this property by assigning an instance of the **iGRowPattern** class, the reference to the specified **iGRowPattern** object isn't stored, but the properties of this object are used to change the corresponding row properties. When you change the properties of the object returned by the **Pattern** property, nothing is changed in the row.

Note also that the row pattern does not store the cells. So the code statement above will not affect cells.

4.5. Moving Rows

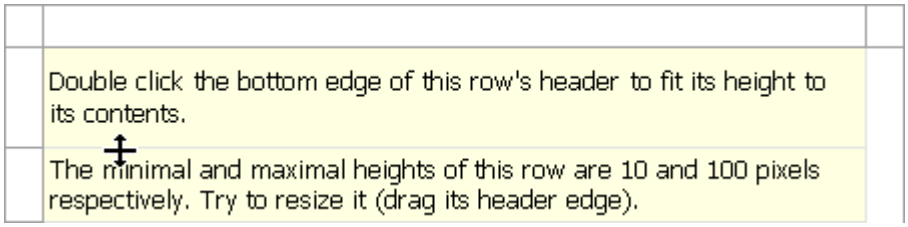
To move rows, use the **Move** method of the **iGRow** class. The only restriction is that rows cannot be moved in the "stay-sorted" mode (see the **StaySorted** property).

Be careful when moving a row within a tree-like grid. iGrid does not automatically change the **VisibleParentExpanded** property. It means than if you move the row from a collapsed group to an expanded one, the row will not become visible. To make it visible, you should collapse and expand the destination group.

4.6. Resizing Rows

The user can change the height of a row by dragging the row's bottom edge. By default, this can be done only in the row header area if it is visible (see the **RowHeader**.**Visible** property). The **RowResizeMode** property allows you to specify areas from which the user can resize rows: in the row header, the frozen columns and row header, or all the columns and the row header.

When the mouse pointer is in the row resize area, it changes to a two-directional arrow indicating the ability to resize the row:



The distance from the bottom row edge in which row resizing is possible can be adjusted with the **RowResizeAreaExtentToDivider** property of iGrid. The related property, **RowResizeAreaProtectedSpace**, can be used to specify the height of the part of the row in which resizing is not possible — to provide the ability to click the row header if it is small enough.

To change the height of a row from code, use the **Height** property of the corresponding row object (**iGRow**).

iGrid raises the **RowHeightStartChange**, **RowHeightChanging**, and **RowHeightEndChange** events before, during, and after the user changes the height of a row.

The **RequestRowResize** event allows you to prohibit the resizing of a row and/or specify the minimal and maximal height for it. If you want to prohibit resizing of a separate row, set the **AllowResizing** parameter of this event to True. To specify the minimal or maximal height of a row, use the **MinHeight** or **MaxHeight** parameter respectively.

The following example demonstrates how to prohibit resizing of group rows and limit the minimal and maximal height of normal rows to one and three rows of text respectively:

```
void iGrid1_RequestRowResize(object sender, iGRequestRowResizeEventArgs e)
{
  if (iGrid1.Rows[e.RowIndex].Type == iGRowType.AutoGroupRow)
  {
    e.AllowResizing = false;
  }
  else
  {
    int myExtra =
      iGrid1.DefaultCol.CellStyle.ContentIndent.Top +
      iGrid1.DefaultCol.CellStyle.ContentIndent.Bottom +
      iGrid1.GridLines.Horizontal.Width;

    e.MinHeight = iGrid1.Font.Height + myExtra;
    e.MaxHeight = iGrid1.Font.Height * 3 + myExtra;
  }
}
```

The **ImmediateRowResizing** property of iGrid determines whether iGrid should update the row contents after every mouse move event while the user is resizing this row. The default value of this property is True, which may be undesirable if the cells contain heavy-to-draw content. To turn repainting off during interactive row resizing, set **ImmediateRowResizing** to False.

4.7. Automatic Adjustment of Row Height

iGrid allows you to adjust the height of a row automatically so that the contents of its cells will be displayed without clipping. This can be done interactively and from code.

To automatically adjust the height of a row interactively, double-click the bottom edge of the row's header. You can also enable the ability to resize the row by dragging its bottoms edge (see the Resizing Rows topic for more information). In this case double-clicking the row's bottom edge performs the same automatic adjustment of the row height.

To automatically adjust the height of a row from code, call the **AutoHeight** method of the corresponding row object (**iGRow**). The row collection object returned by the **iGrid**.**Rows** property also implements its own **AutoHeight** method. Use it to automatically adjust the height of all rows in one call.

When iGrid automatically adjust row heights, it takes into account the text format flags of the cells. By default a cell does not use word wrapping (the **TextFormatFlags** property does not contain the **WordWrap** flag), and iGrid resizes the row to display all text lines of the cell while some text lines can be clipped if the width of the cell is not enough to display them. If the cell uses word wrapping, the required row height will be calculated taking into account this flag — all the text lines of the cell will be wrapped using the current width of the cell and as the result the cell's text will be fully visible.

If you want to specify the minimal and/or maximal height for one or more rows when performing an automatic height adjustment operation, use the **MinHeight** and **MaxHeight** event properties of the **RequestRowResize** event. Below is an example of how to limit the height of all rows to a range of 10 to 50 pixels:

```
private void iGrid1_RequestRowResize(
  object sender, iGRequestRowResizeEventArgs e)
{
  e.MinHeight = 10;
  e.MaxHeight = 50;
}
```

The **GetPreferredRowHeight** method of iGrid allows you to calculate the minimum row height needed to display the contents of a row without adding real rows to the grid. This method calculates the row height based on the column properties. The **hasText** and **hasImage** parameters of this method allow you to specify whether the rows will have images and text respectively. Invoke this method after the columns are added and adjusted, but before adding rows. The following example shows how to use this method:

```
...
//add the columns
...
iGrid1.DefaultRow.Height = iGrid1.GetPreferredRowHeight(true, true);
...
//add the rows
...
```

Some notes about automatic row height adjustment:

- When you automatically adjust a row's height, iGrid can take into account the row's header to display its contents without clipping (the default behavior). You can change this mode with the **AutoHeightRowMode** property which accepts one of the following values: **Cells**, **RowHdr**, and **RowHdrAndCells**.
- iGrid automatically fits a row's height when the user double-clicks the edge of the row. To prohibit this behavior, use the **RowDividerDoubleClick** event (the **DoDefault** property of the event arguments).
- iGrid supports the "auto-height" operation for custom-drawn cells and row headers too. You should handle the **CustomDrawCellGetHeight** and **CustomDrawRowHdrGetHeight** events to provide the height of your custom-drawn contents for this operation.
- When the **GetPreferredRowHeight** method is invoked, the **CustomDrawCellGetHeight** and **CustomDrawRowHdrGetHeight** events may be raised and the **RowIndex** argument of these events will be equal to -1.

If iGrid contains merged cells, you can control how they are processed during automatic row height adjustment with the help of the **AutoHeightRowSpanHandling** property of iGrid. This capability

is described in greater detail in the Merged Cells in Auto-sizing Operations topic in the section devoted to merged cells.

4.8. Alternating Row Colors

The **BackColorEvenRows** and **BackColorOddRows** properties of iGrid allow you to set different background colors for even and odd rows without changing the background color of every row.

The styles of a column, row, and cell have superiority over these properties. So if you set the background color of even rows to red and at the same time a cell in an even row will have a style with **BackColor** set to blue, the cell will have the blue background.

4.9. Row Visibility System

The **iGRow** class representing an iGrid row implements three Boolean properties related to row visibility: **Visible**, **VisibleFiltered**, and **VisibleParentExpanded**. Each of these properties determines whether a row will be visible in the corresponding context. And only if all these three properties are True, the row will be rendered on the screen. Let's consider what each property is responsible for.

If you as a developer want to make some rows always invisible for the user, you should use the **iGRow**.**Visible** property. Set it to False for the corresponding rows to hide them. In the vast majority of cases you will assign values only to this row visibility property out of the three listed above to change row visibility.

If you enable the search-as-type functionality in filter mode or attach the AutoFilterManager add-on to provide more sophisticated filtering in iGrid, these tools change another row property to make rows visible depending on the user's filter — **VisibleFiltered**. You can check the value of this property after applying a filter to know whether a row meets the filter criteria (True — yes, False — no). In the vast majority of applications this property will never be changed by you — unless you will decide to implement your own filtering system as a replacement for AutoFilterManager or the built-in search-as-type window.

What is very important in the context of user filtering with the built-in search-as-type functionality or the AutoFilterManager add-on is that they ignore the rows you as a developer made invisible by setting their **Visible** property to False. This brings you the ability to implement your own filters and combine them with search-as-type or autofilter filtering. Your filters can be quite complex so they can't be constructed with AutoFilterManager, or vice versa. You can make some rows invisible based on an algorithm or a query to a database by setting their **Visible** property to False and provide your users with the ability to filter them interactively with AutoFilterManager.

The last of the three row visibility properties is **VisibleParentExpanded**. The functionality of this property is similar to **VisibleFiltered**, but it is used to make rows invisible in tree grids or grids with group rows. When the user clicks the plus/minus button in a group row or tree node to collapse it, all its child rows must become temporarily invisible — until the user will expand that row again. To hide child rows, iGrid sets their **VisibleParentExpanded** property to False. If a row should become visible because its parents become expanded, **VisibleParentExpanded** is set to True. You will change this property from code only in some specific scenarios — for example, when you insert new rows into a tree grid after it has been originally populated or move rows from one group to another.

These row visibility properties are also considered in other parts of this manual in which the corresponding functionality is described in greater detail. The last thing regarding these three properties that is worth to be mentioned here is that they all are set to True by default for every new row.

5. CELL FEATURES

5.1. Cell Structure and Cell Styles

Every iGrid cell is represented with a data structure in memory. An instance of this structure has the following 6 fields:

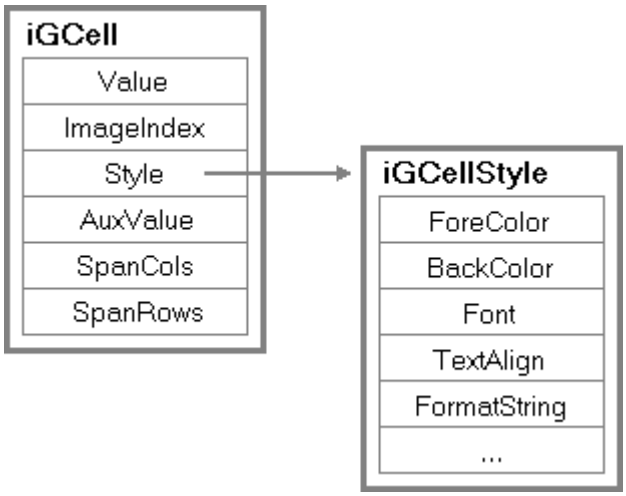
- the cell value;
- the index of the image displayed in the cell;
- the reference to the style object that defines cell formatting (background color, text font, alignment, etc.);
- the auxiliary value used by iGrid when a drop-down list is attached to the cell or for your own needs in other cases;
- the number of columns the cell spans over;
- the number of rows the cell spans over.

When you read or write properties of an iGrid cell, you deal with an instance of the **iGCell** class. You retrieve a reference to an **iGCell** object that represents a cell when you access grid cells with the **Cells** collection of iGrid, for instance:

```
iGCell myCell = iGrid1.Cells[0, 3];
myCell.Value = 123;
myCell.ImageIndex = 1;
```

The 6 constituent parts of the iGrid cell listed above are represented with the following properties of the **iGCell** class: **Value**, **ImageIndex**, **Style**, **AuxValue**, **SpanCols**, **SpanRows**. The **Value** and **AuxValue** properties are of the **Object** type; **ImageIndex**, **SpanCols** and **SpanRows** are of the **Int32** type; the **Style** property contains a reference to a cell style object of the **iGCellStyle** type.

But if you look at the full list of the properties provided by the **iGCell** class, you will see much more than these 6 basic properties. Other cell properties, such as **BackColor**, define the formatting and behavior of the cell. All calls to these properties from your code are translated into calls to the corresponding properties of the style object attached to the cell. The following schema illustrates the relation between the cell data and its style object:



By default, the **Style** property of a cell is not initialized (null in C#, Nothing in VB.NET). If you assign a value to one of the properties from the cell-style group, a new cell style object is automatically created, its corresponding property is set to the value you assign, and a reference to this cell style object is stored in the **Style** property of the cell.

As you see, you can format individual cells directly through their formatting properties or by creating a cell style object with the required formatting and assigning it to the cell. Both approaches have

practically the same performance, and the approach you will use depends on a particular task. For instance, if you need to format some cells using different formatting parameters, the former method with accessing cell properties directly is more preferable. In another scenario, if you need to apply the same formatting to several cells, the latter method with cell styles will require less code and will save memory. The latter method is also much better when you need to change the formatting of a group of similar cells. In this case you can simply change the formatting in the cell style object used to format those cells.

To complete the explanation, here is an example of how you can format cells through their properties:

```
iGrid1.Cells[0, 0].BackColor = Color.Red;
iGrid1.Cells[0, 0].ForeColor = Color.Yellow;
iGrid1.Cells[0, 0].TextAlign = iGContentAlignment.TopCenter;

iGrid1.Cells[1, 0].BackColor = Color.Red;
iGrid1.Cells[1, 0].ForeColor = Color.Yellow;
iGrid1.Cells[1, 0].TextAlign = iGContentAlignment.TopCenter;

iGrid1.Cells[2, 0].BackColor = Color.Red;
iGrid1.Cells[2, 0].ForeColor = Color.Yellow;
iGrid1.Cells[2, 0].TextAlign = iGContentAlignment.TopCenter;
```

And below you see an equivalent code based on cell styles:

```
iGCellStyle myStyle = new iGCellStyle();

myStyle.BackColor = Color.Red;
myStyle.ForeColor = Color.Yellow;
myStyle.TextAlign = iGContentAlignment.TopCenter;

iGrid1.Cells[0, 0].Style = myStyle;
iGrid1.Cells[1, 0].Style = myStyle;
iGrid1.Cells[2, 0].Style = myStyle;
```

The only difference is that in the first snippet each cell has its own cell style object created automatically, while in the second snippet the cells reference the same cell style. The same cell style object will allow you to change the text color or another formatting option for all those cells in one statement like this:

```
myStyle.ForeColor = Color.Black;
```

5.2. Cell Style Inheritance

The effective values of an iGrid cell style object determine the look and behavior of cells this style is applied to. This means that the cell style object must provide some defined values for all its properties. However, when a new cell style object is created, all its properties are non-initialized. So where do certain property values come from? The answer is the cell style inheritance model iGrid is based on.

When a new style object is created, all its properties are set to special "NotSet" values meaning that values of the style properties are inherited from the styles up the hierarchy. The cell's style has the lowest priority, then the row cell style and column cell style go (in this order). The highest priority have so-called super values determined by the grid properties or special hard-coded constants.

NotSet values depend on the property type. For object types it is null (Nothing in VB), for color properties — **Color**.**Empty**, for enum properties — the NotSet value from the enum, and so on.

As an example, let's consider how the foreground color of a cell is determined. First iGrid checks whether a style is attached to the **Style** property of the cell. If a style is attached, iGrid checks whether the **ForeColor** property of this style has a value (is not set to **Empty**). If so, this color is used. Otherwise iGrid checks whether a style is attached to the **CellStyle** property of the row that contains the cell (see the **iGRow**.**CellStyle** property). If a style is attached, and its **ForeColor** property is not empty, this color is used. Otherwise the **ForeColor** of the style attached to the column containing the cell (**iGCol**.**CellStyle**) is used.

Note that column cell style properties can also be set to NotSet values. If iGrid has a property that can be used as the top-level setting for the property (**ForeColor**, **BackColor**, **Font**), its value is used. If there is no corresponding property, a hard-coded default value is used. For instance, the NotSet value in the **Type** property defaults to the text box cell type if no corresponding property is set for the column cell style object.

The following table lists hard-coded default values for cell properties if they can't be determined from the cell style hierarchy. Only non-empty default values are listed for brevity:

PROPERTY	DEFAULT VALUE
ContentIndent	An **iGIndent** structure with all fields set to 1.
Flags	A combination of the **iGCellFlags**.**DisplayImage** and **iGCellFlags**.**DisplayText** flags.
EmptyStringAs	**iGEmptyStringAs**.**Null**
TextAlign	**iGContentAlignment**.**MiddleLeft**
TextPosToImage	**iGTextPosToImage**.**Horizontally**
Type	**iGCellType**.**Text**
TextTrimming	**iGStringTrimming**.**EllipsisCharacter**

Let's consider an example demonstrating inheritance in cell style property value. For example, we have a column named "Price", and we want to set the background color of all the cells in the column to light gray. We can do this through the column's cell style object:

```
iGrid1.Cols["Price"].CellStyle.BackColor = Color.LightGray;
```

In addition we want to mark all the cells with values greater than 10 with red. We can do this as follows:

```
foreach(iGCell cell in iGrid1.Cols["Price"].Cells)
{
  if ((int)cell.Value > 10)
  {
    cell.BackColor = Color.Red;
    //this operation is equivalent to:
    //cell.Style = new iGCellStyle();
    //cell.Style.BackColor = Color.Red;
  }
}
```

A better version of this code, which does not create a cell style object for every cell with a value greater than 10, is shown below

```
iGCellStyle highlightStyle = new iGCellStyle();
highlightStyle.BackColor = Color.Red;

foreach(iGCell cell in iGrid1.Cols["Price"].Cells)
{
  if ((int)cell.Value >= 10)
    cell.Style = highlightStyle;
}
```

In the first example each cell with a value greater than 10 will have own style. All the properties of this style object are set to the NotSet values, except **BackColor**, which is set to **Red**. So the cell with this style object will have the font, foreground color, text alignment, ... defined in the column's cell style object and own background color (red).

In the second example we created the required cell style object beforehand and applied it to the cells to be highlighted. Later, if we want to change the background color of all highlighted cells from red to green, we can simply do the following:

```
highlightStyle.BackColor = Color.Green;
```

And all the cells to which were applied the highlightStyle style will change the background color to green.

5.3. Using Cell Styles at Design Time

Actually there are two classes that represent a cell style in iGrid: **iGCellStyle** and **iGCellStyleDesign**. The main difference between them is that only **iGCellStyleDesign** is visible at design time in the Windows Forms Designer. It also provides some design-time features that are not needed in **iGCellStyle** objects used solely from code.

The **iGCellStyleDesign** class provides the ability to add it to the Visual Studio Toolbox. You can create instances of this class at design time by dragging the **iGCellStyleDesign** item from the Toolbox to a Windows form. Some cell style properties of iGrid objects also allow you to create **iGCellStyleDesign** objects by choosing the 'Create new ...' command while editing these properties in the property editor.

Appearance of a column header (header cell) can also be specified with two styles: **iGColHdrStyle** and **iGColHdrStyleDesign**. The difference between them is the same as between **iGCellStyle** and **iGCellStyleDesign**.

5.4. Cell Appearance and Behavior Properties

The iGrid cell style object (**iGCellStyle**) implements the following properties that define the look and behavior of an iGrid cell:

PROPERTY	DESCRIPTION
BackColor	The background color of the cell.
ContentIndent	Defines the left, top, right and bottom indent of the cell contents.
CustomDrawFlags	Defines which layers of the cell (background, foreground) will use custom drawing.
CustomEditor	Gets or sets the object used as the custom cell editor.

PROPERTY	DESCRIPTION
DropDownControl	Gets or sets the control to show as the drop-down editor of the cell. As a rule, it is used to create combo box cells.
EmptyStringAs	Specifies how to interpret an empty string entered into the cell.
Enabled	Determines whether the cell is enabled for editing or not. Disabled cells use the **ForeColorDisabled** color to draw its text and cannot be edited.
FitContentsInViewport	Indicates whether iGrid must reallocate the cell contents to draw them in the viewport area.
Flags	Gets or sets flags that determine which parts of the cell's contents (image, text) are displayed.
Font	Gets or sets the font of the text displayed in the cell.
ForeColor	The foreground color of the cell (the cell text color).
FormatProvider	Contains a reference to an object implementing the **IFormatProvider** interface (to implement complex formatting when the **FormatString** property is not enough).
FormatString	Defines the format string applied to the cell value before it is displayed on the screen (like in the .NET **String**.**Format** method).
ImageAlign	The horizontal and vertical alignment of the image in the cell.
ImageList	Gets or sets the image list that contains the images to display in the cell.
MaxInputLength	Gets or sets the maximum number of characters that can be entered into the cell.
ReadOnly	Gets or sets a value indicating whether the cell can be edited.
Selectable	Determines whether the cell can be selected with the mouse and/or keyboard.
SingleClickEdit	Gets or sets a value indicating whether the single mouse click on a cell starts editing of the cell if it is not current.
TextAlign	The horizontal and vertical alignment of the cell text.
TextFormatFlags	Defines the display and layout information for cell text (like the **StringFormatFlags** enumeration in the .NET Framework).
TextPosToImage	The relative position of the cell icon and the cell text.
TextTrimming	Defines the cell text trimming options (like the **StringTrimming** enumeration in the .NET Framework).

PROPERTY	DESCRIPTION
Type	Specifies the type of the cell (text box or check box cell).
TypeFlags	Additional flags to modify cell functionality depending on the cell type.
ValueType	Gets or sets the type of value to which iGrid should convert the text entered while editing.

5.5. Formatting Cells

5.5.1. Cell Text Formatting Options

An iGrid cell provides several properties allowing you to control how the cell text is rendered on the screen.

The first of them is the **TextAlign** property that specifies the horizontal and vertical alignment of the cell text. For example, you can set the horizontal right alignment and center cell text of the very first cell with the following statement:

```
iGrid1.Cells[0, 0].TextAlign = iGContentAlignment.MiddleRight;
```

The next property you can use to control the cell text rendering process is **TextFormatFlags**. It specifies the display and layout information for cell texts and has the **iGStringFormatFlags** enumeration type similar to the **StringFormatFlags** enumeration in the .NET Framework. One of the most commonly used features provided by this property is the creation of cells with multiline text. This feature is enabled with the **WordWrap** flag, for example:

```
iGrid1.Cells[0, 0].TextFormatFlags = iGStringFormatFlags.WordWrap;
```

The **TextFormatFlags** property is a flag property accepting values from the **iGStringFormatFlags** enumeration. As such, you can combine several flags to achieve the desired effect. For example, you can make cell text multiline and vertical with the following setting:

```
iGrid1.Cells[0, 0].TextFormatFlags =
  iGStringFormatFlags.WordWrap | iGStringFormatFlags.DirectionVertical;
```

To control how the cell text is trimmed if the cell does not have enough space to display the cell text in its entirety, use the **TextTrimming** property. This property accepts values from the **iGStringTrimming** enumeration similar to the **StringTrimming** enumeration in the .NET Framework. The default cell text trimming option is **EllipsisCharacter**, but you can change it, for example, to **EllipsisPath** that works better for cell texts displaying paths to folders:

```
iGrid1.Cells[0, 0].TextTrimming = iGStringTrimming.EllipsisPath;
```

5.5.2. Coloring Cells

When iGrid determines the background color of a cell, it checks the values of the following properties in the order listed below:

1. The **BackColor** property of the cell, which is the background color of its style (**iGCell**.**Style**).
2. The **BackColor** property of the cell style of the row (**iGRow**.**CellStyle**).
3. The **BackColor** property of the cell style of the column (**iGCol**.**CellStyle**).
4. The **BackColor** property of the grid.

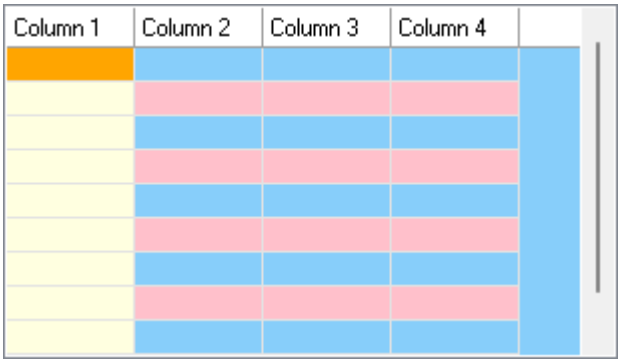
The first found non-empty color value is used to fill the background of the cell.

You can also specify the background color of the even and odd rows with the two iGrid properties — **BackColorEvenRows** and **BackColorOddRows**. If a background color is assigned to the style of a cell, its column or row, it takes precedence over the colors specified in the **BackColorEvenRows** and **BackColorOddRows** properties.

The following example shows how to set up the background color of a grid, its even rows, a column, and a cell:

```
//Set the background color of the grid
iGrid1.BackColor = Color.LightSkyBlue;

//Set the background color of the even rows
iGrid1.BackColorEvenRows = Color.Pink;

//Set the background color of the first column
iGrid1.Cols[0].CellStyle.BackColor = Color.LightYellow;

//Set the background color of the first cell
iGrid1.Cells[0, 0].BackColor = Color.Orange;

//The last statement is equivalent to:
//iGCellStyle myStyle = new iGCellStyle();
//myStyle.BackColor = Color.Orange;
//iGrid1.Cells[0, 0].Style = myStyle;
```

The result is on the picture below:



All the principles outlined above also work for the foreground color of the iGrid cell. The equivalent **ForeColor** properties are used for that (such as **iGCell**.**ForeColor**).

You can also color cells dynamically using the **CellDynamicFormatting** event of iGrid. For more information, read the Formatting Cells with Events topic.

The background and foreground color of the cell do not affect the colors of the elementary controls in it (combo and ellipsis buttons, check box, tree button). The colors of these cell elements are specified by the **CellCtrlBackColor**, **CellCtrlForeColor**, and **CellCtrlForeColorDisabled** properties of iGrid. The ability to customize the colors of controls displayed inside cells also depends on the rendering style. Read the Built-in Rendering Styles topic to find out more.

5.5.3. Cell Format Strings

Cell values are stored in their native format and are displayed "as is" by default. But in many cases it is more convenient for the user to perceive cell values if they are presented in a special format. This is especially useful for date or currency values. For example, the user would definitely prefer to see the dollar amount of 1234567.89 as "$1,234,567.89". The **FormatString** property of the iGrid cell or its style is used for this purpose.

The **FormatString** property uses the same syntax as the **format** parameter in the .NET **String**.**Format** method. If a format string is specified in the **FormatString** property, actually iGrid uses a call like **String.Format(string format, object arg0)** to get the text representation of the cell value on the screen. The **format** parameter is set to the string specified in the **FormatString** property of the cell, the **arg0** parameter is set to the cell value in this call.

For example, we can set the format string of a cell to "{0:C}", where "C" is the default currency format in the current culture. If the cell value is 1234567.89, the user will see "$1,234,567.89" in the cell on the screen in the en-US culture — because **String.Format("{0:C}", 1234567.89)** returns the same result.

"{0}" in the cell format string refers to the cell value. In the general case you can refer to the cell value several times in a format string. The following example demonstrates how to separate a **DateTime** value into the date and time parts in the cells of the second column:

```
iGrid1.Cols[1].CellStyle.FormatString = "Date — {0:d}, time — {0:HH:MM}";
```

For example, if a cell in the second column of such a grid will contain the date value created with the expression **new DateTime(2023, 3, 5, 11, 30, 15)**, the user will see "Date — 3/5/2023, time — 11:30" in this cell in the en-US culture.

Pay attention to the fact that formatting of values is performed according to the current culture of the application (if this is applicable to a particular format string). In some cases it is good because iGrid automatically adjusts the format of numeric and date values according to the user's culture. In other cases this may be undesirable — for example, if you want to use a specific culture regardless of the user's one. Format providers can help you to implement this and other tasks. For more information about format providers, read the Cell Format Providers topic.

If a cell value was formatted with a format string, you can retrieve the cell text the user sees on the screen with the **Text** property of the corresponding **iGCell** object.

5.5.4. Cell Format Providers

The Cell Format Strings topic describes how you can format raw cell values using .NET format strings. The formatting is performed according to the current culture of the application. This is good if you want the application to format numeric and date values according to the user's culture, but may be undesirable in other situations — for example, if you always want to use the formatting rules of a specific culture. In some rare cases you may need a more complex formatting algorithm that cannot be implemented using custom format strings. Format providers can help you in both situations.

A format provider is an object implementing the .NET **IFormatProvider** interface. You can assign an object implementing this interface to the **FormatProvider** property of the cell. In this case iGrid will call the **String.Format(IFormatProvider provider, string format, object arg0)** method to get the string representation of the cell value on the screen. The following table lists the parameters passed to the call of the **String**.**Format** method in this case:

PARAMETER	VALUE
provider	The value of the **FormatProvider** property of the cell.
format	The value of the **FormatString** property of the cell.
arg0	The value of the **Value** property of the cell.

Below you will find two examples demonstrating how to implement formatting of iGrid cell values using a specific culture and how to implement a custom format provider. More information about format providers in .NET can be found in the documentation for the **String**.**Format** method.

Using specific culture for formatting

To format cell values using a specific culture, you need an instance of the **CultureInfo** object from the .NET **System.Globalization** namespace. Its **NumberFormat** property returns a **NumberFormatInfo** object that defines the culturally appropriate format of displaying numbers, currency, and percentage. This object implements the **IFormatProvider** interface and can be used in iGrid to format numeric values using the specific culture.

For example, if we want to format a numeric value to display it with thousand separators using the traditional Ukrainian thousand separator character, we can use a code snippet like this:

```
CultureInfo culture = new CultureInfo("uk-UA");
iGrid1.Cells[2, 0].FormatProvider = culture.NumberFormat;
iGrid1.Cells[2, 0].FormatString = "{0:#,##}";
```

The **DateTimeFormatInfo** object returned by the **DateTimeFormat** property of the **CultureInfo** object can be used for the same purpose to format **DateTime** values.

Implementing custom format provider

If you need a complex formatting algorithm that cannot be implemented using custom format strings, you can create a custom format provider based on the .NET **IFormatProvider** interface and use it to format iGrid cells. To do that, specify your own format string in the **FormatString** property of the cell, and assign a reference to the corresponding custom format provider to the **FormatProvider** property of the cell.

The example below demonstrates how to create a format provider for IP addresses stored as Int64 values. This format provider is used to format the cells in the first column:

```
// specify own "ip" format
iGrid1.Cols[0].CellStyle.FormatString = "{0:ip}";
// set the format provider for the IP Address column
iGrid1.Cols[0].CellStyle.FormatProvider = new IPFormatProvider();
...
public class IPFormatProvider : IFormatProvider, ICustomFormatter
{
  public object GetFormat(Type service)
  {
    if (service == typeof (ICustomFormatter))
      return this;
    else
      return null;
  }

  public string Format(
    string format, object arg, IFormatProvider provider)
  {
    if (format == null)
      return arg.ToString();

    if (!format.Trim().Equals("ip"))
    {
      if (arg is IFormattable)
      {
        return ((IFormattable)arg).ToString(format, provider);
      }
      else if (arg != null)
      {
        return arg.ToString();
      }
      return string.Empty;
    }

    uint myLong = (uint)arg;

    return string.Format(
      "{0:000}.{1:000}.{2:000}.{3:000}",
      (myLong>>24 & 0xff),
      (myLong>>16 & 0xff),
      (myLong>>8 & 0xff),
      (myLong & 0xff));
  }
}
```

Pay attention to the fact that the **FormatString** property of the cell style used to format the cells in the first column contains the special "ip" identifier of the custom format provider implemented in the IPFormatProvider class.

5.5.5. Formatting Cells with Events

iGrid also allows you to format cells with events. When iGrid is going to draw a cell, first it determines the appearance parameters from the cell properties (see the Coloring Cells topic). After that iGrid raises the following events to give you the ability to change these parameters if required:

EVENT	DESCRIPTION
CellDynamicFormatting	Allows you to dynamically determine the background and foreground colors and the font of the cell just before drawing.
CellDynamicStringFormat	Allows you to dynamically determine a **StringFormat** object used by iGrid to format the cell value just before drawing.

The arguments of each event contain the row/column indexes of the cell and the fields that define the cell appearance (font, background color, etc.). The values of these fields are initialized with values calculated from the cell properties, but you can change them in handlers of these events. For example, if you want to highlight rows with values in a Price column greater than 10, you can do as follows:

```
private void iGrid1_CellDynamicFormatting(
  object sender, iGCellDynamicFormattingEventArgs e)
{
  if ((int)iGrid1.Cells[e.RowIndex, "Price"].Value > 10)
    e.BackColor = Color.Red;
}
```

5.6. Displaying Images in Cells

An iGrid cell can display text, an image, or both. In this topic, we will focus on the iGrid ability to display images in cells.

The first step in displaying images in iGrid cells is to provide a source of images. The standard way to provide images for WinForms controls is to use the built-in .NET **ImageList** component, and iGrid fully supports this technique as well. If you have an ImageList, you can specify it as the image source for iGrid by assigning a reference to this ImageList to the **ImageList** property of iGrid.

Let's suppose you have a form with iGrid named iGrid1 on it. You can add an ImageList component to the form and upload images you want to display in iGrid into it at design time. If you leave the default ImageList name imageList1 unchanged, the following statement will be used to set this ImageList as the image source for our iGrid:

```
iGrid1.ImageList = imageList1;
```

To display an image from an ImageList in an iGrid cell, we should specify the index of the image in the **ImageIndex** property of the cell. The first image has the index of 0, the second image has the index of 1, and so on. The default value of the **ImageIndex** property of any iGrid cell is -1, which means no image will be displayed in the cell.

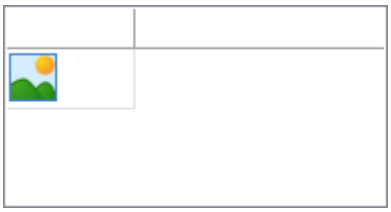
Let's continue building our example and show how to create a cell in which we will display the first image from the ImageList imageList1. We can create one column and one row with the following code:

```
iGrid1.Cols.Count = 1;
iGrid1.Rows.Count = 1;
iGrid1.Rows[0].Height = 30;
```

And after that set the image index in the cell to 0 to display the first image from imageList1 we specified as iGrid1's image source above:

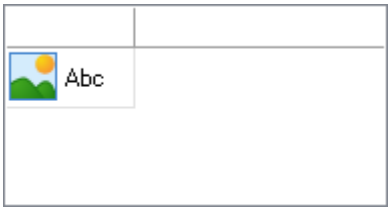
```
iGrid1.Cells[0, 0].ImageIndex = 0;
```

You can see the result on the screenshot below:



Let's add some text to the cell to display it together with the image:

```
iGrid1.Cells[0, 0].Value = "Abc";
```



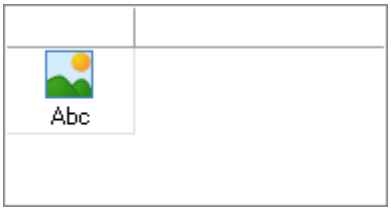
As you see, the cell text is displayed at the right of the cell image. We can change this default layout with the **TextPosToImage** property of the cell. This property accepts values from the **iGTextPosToImage** enumeration allowing us to place the cell text next to the cell image in the horizontal or vertical direction, or even place the cell text over the cell image.

Let's center the image and text in our cell horizontally, but now place the text below the image. Two other iGrid cell properties, **ImageAlign** and **TextAlign**, will help us to specify the desired alignment for the image and text elements:

```
iGrid1.Cells[0, 0].TextPosToImage = iGTextPosToImage.Vertically;
iGrid1.Cells[0, 0].ImageAlign = iGContentAlignment.TopCenter;
iGrid1.Cells[0, 0].TextAlign = iGContentAlignment.BottomCenter;
```

Let's also automatically adjust the height of the row with the **AutoHeight** method to display the cell contents without clipping:

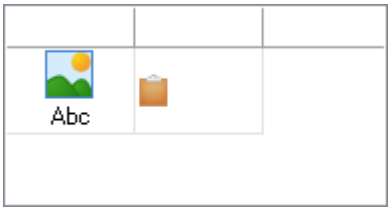
```
iGrid1.Rows[0].AutoHeight();
```



Another way to specify an image source for cells is the **ImageList** property of an iGrid cell or a style object attached to a cell. In the general case every iGrid cell can have its own image source with different property values. This means that you can display images of different sizes and color depths inside the same iGrid control. As an example of this, let's add a cell to our iGrid1 and display an image of a smaller size from another ImageList named imageList2:

```
iGrid1.Cols.Add();
iGrid1.Cells[0, 1].ImageList = imageList2;
iGrid1.Cells[0, 1].ImageIndex = 1;
```

The result may look like this:



If you want to remove an image from a cell, you can set the **ImageIndex** property of the cell to its default value of -1. If you want to temporarily hide the cell image without setting the cell image index to -1, use the **Flags** property of the cell. This property accepts a combination of flags indicating which parts of a cell, image and/or text, are displayed. By default both cell parts are displayed. If you want to hide the image and display only text, execute a statement like this:

```
iGrid1.Cells[0, 0].Flags = iGCellFlags.DisplayText;
```

In some scenarios you may need to display the same image in all cells of a column. iGrid allows you to implement this task easily using the **DefaultCellImageIndex** property of the column. For example, you can specify that all new cells in the second column will display the image with index 3 with a statement like the following one:

```
iGrid1.Cols[1].DefaultCellImageIndex = 3;
```

Note that this statement must be executed before you create rows in the grid control.

In all ways of displaying an image in a cell described above you specify an image index statically. In iGrid, you can also provide an image for a cell dynamically based on the cell value or other rules. This task can be implemented with the **CellDynamicContents** event of iGrid. The Cell Dynamic Contents topic explains this technique.

5.7. Cell Dynamic Contents

The value of an iGrid cell is something stored in the computer memory. To show it to the user on the screen, iGrid must retrieve the text representation of the value and render it with .NET string output methods. In the simplest case iGrid applies the universal **ToString** method to the cell value to get its text representation. If a format string is specified, iGrid uses it to retrieve the cell text. You can find out more about this process from the Cell Format Strings topic.

iGrid provides one more tool you can use to tell iGrid the text and/or image to output in the cell. This feature is called "cell dynamic contents". It can be used to adjust the cell text retrieved with the methods described in the previous paragraph, or to obtain cell text from other sources — for example, to implement a kind of a virtual grid.

To provide cell text dynamically, create an event handler for the **CellDynamicContents** event of iGrid and set the cell text in the **Text** property of the event arguments object. As an example, below is the source code of the form in which iGrid displays words "One", "Two", "Three" instead of the corresponding integer values:

```
private void Form1_Load(object sender, EventArgs e)
{
  iGrid1.Cols.Count = 1;
  iGrid1.Rows.Count = 3;

  iGrid1.CellValues[0, 0] = 1;
  iGrid1.CellValues[1, 0] = 2;
  iGrid1.CellValues[2, 0] = 3;

  iGrid1.CellDynamicContents += iGrid1_CellDynamicContents;
}

private void iGrid1_CellDynamicContents(
  object sender, iGCellDynamicContentsEventArgs e)
{
  int value = (int)iGrid1.CellValues[e.RowIndex, e.ColIndex];
  switch (value)
  {
    case 1:
      e.Text = "One";
      break;
    case 2:
      e.Text = "Two";
      break;
    case 3:
      e.Text = "Three";
      break;
  }
}
```

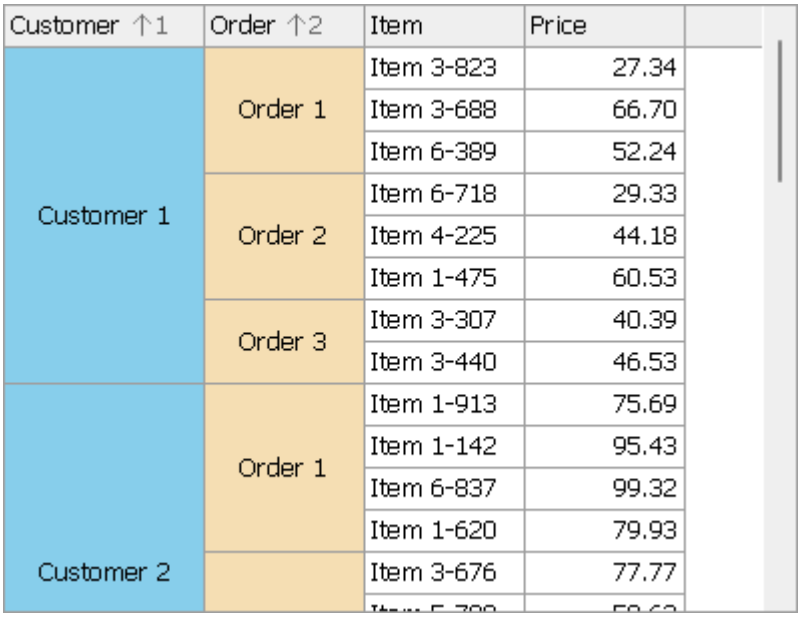
When iGrid raises this event, the **Text** property of its event arguments object already contains the cell text retrieved with one of the methods described in the first paragraph of this topic. You can leave it "as is" or modify for some cells if required. The other properties of the **CellDynamicContents** event arguments object, **ImageIndex** and **CheckState**, allow you to specify dynamically the cell image index and check state for check box cells the same way.

If you need to retrieve the cell text the user sees on the screen, you can do that with the **Text** property of the cell.

5.8. Forcing iGrid to Draw Cell Contents in Viewport

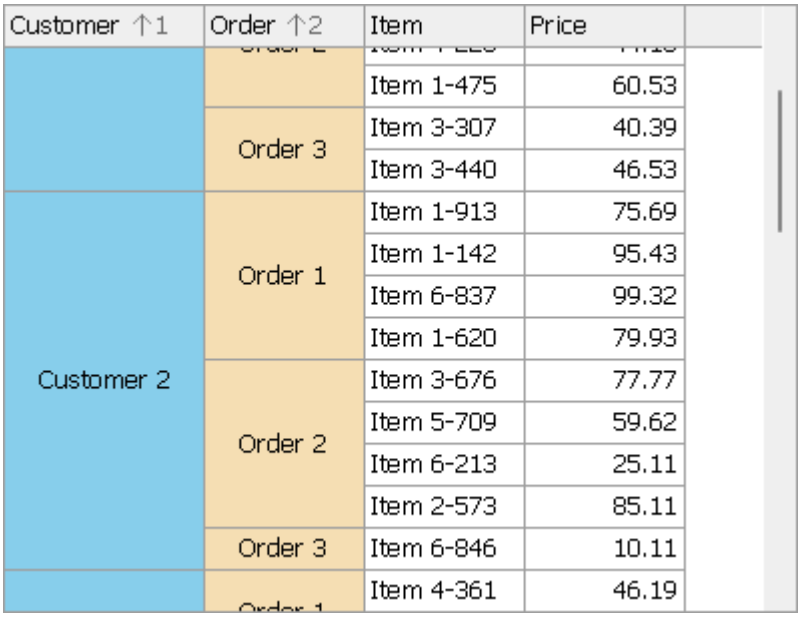
You can force iGrid to draw cell contents in the cell area viewport. This feature is especially useful for merged cells with heights or widths exceeding the size of the visible cell area.

As an example, let's suppose we have the following grid with merged cells created with the help of the **MergeCellsInCols** method:



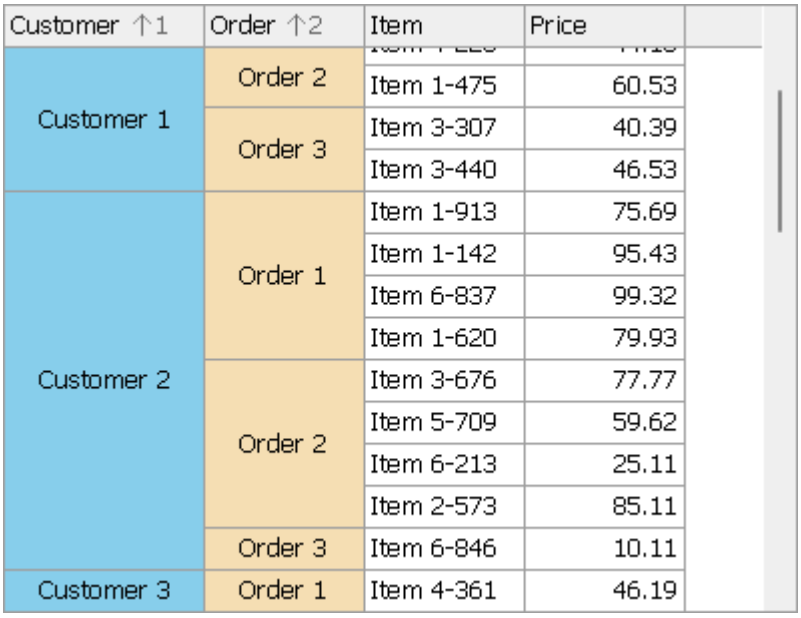
Customer ↑1	Order ↑2	Item	Price
Customer 1	Order 1	Item 3-823	27.34
		Item 3-688	66.70
		Item 6-389	52.24
	Order 2	Item 6-718	29.33
		Item 4-225	44.18
		Item 1-475	60.53
	Order 3	Item 3-307	40.39
		Item 3-440	46.53
Customer 2	Order 1	Item 1-913	75.69
		Item 1-142	95.43
		Item 6-837	99.32
		Item 1-620	79.93
		Item 3-676	77.77

If we scroll this grid in the vertical direction a little bit, we may get the following picture:



Customer ↑1	Order ↑2	Item	Price
		Item 1-475	60.53
	Order 3	Item 3-307	40.39
		Item 3-440	46.53
Customer 2	Order 1	Item 1-913	75.69
		Item 1-142	95.43
		Item 6-837	99.32
		Item 1-620	79.93
	Order 2	Item 3-676	77.77
		Item 5-709	59.62
		Item 6-213	25.11
		Item 2-573	85.11
	Order 3	Item 6-846	10.11
		Item 4-361	46.19

As you can see, now it's not clear for what customer and order we see the item and price in the first visible row below the column header area. To improve the situation, we can force iGrid to reposition the contents of merged cells vertically to draw them in the visible cell area using the **FitContentsInViewport** property of iGrid cells:



Customer ↑1	Order ↑2	Item	Price
Customer 1	Order 2	Item 1-475	60.53
	Order 3	Item 3-307	40.39
		Item 3-440	46.53
Customer 2	Order 1	Item 1-913	75.69
		Item 1-142	95.43
		Item 6-837	99.32
		Item 1-620	79.93
	Order 2	Item 3-676	77.77
		Item 5-709	59.62
		Item 6-213	25.11
		Item 2-573	85.11
	Order 3	Item 6-846	10.11
Customer 3	Order 1	Item 4-361	46.19

This task can be implemented by setting the column cell style's **FitContentsInViewport** property to the **Vertically** value like in the following code:

```
iGrid1.Cols[0].CellStyle.FitContentsInViewport =
  iGCellFitContentsInViewport.Vertically;
```

The cell's **FitContentsInViewport** property and its equivalent property in the cell style object (**iGCellStyle**) accept values form the **iGCellFitContentsInViewport** enumeration. This enumeration is marked with the .NET Flags attribute so you can force iGrid to reallocate cell contents vertically and horizontally by combining the corresponding enumeration values:

```
iGrid1.Cols[0].CellStyle.FitContentsInViewport =
  iGCellFitContentsInViewport.Horizontally |
  iGCellFitContentsInViewport.Vertically;
```

5.9. Custom-Drawn Cells

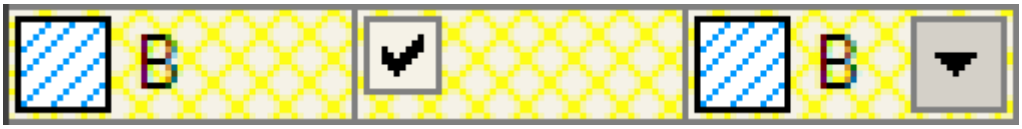
iGrid allows you to use custom drawing in cells. This means you can draw a cell's background, foreground, or both cell parts by yourself. As a rule, custom contents are drawn with GDI+.

To create a custom-drawn cell, do the following:

1. Set the **CustomDrawFlags** property of the cell's style to a combination of the **Background** or **Foreground** values (to specify which parts will be drawn by you).
2. Add an event handler to the **CustomDrawCellBackground** and/or **CustomDrawCellForeground** events.
3. Add an event handler to the **CustomDrawCellGetHeight** and **CustomDrawCellGetWidth** events if you need to tell iGrid how to fit the height and width of the cell automatically.

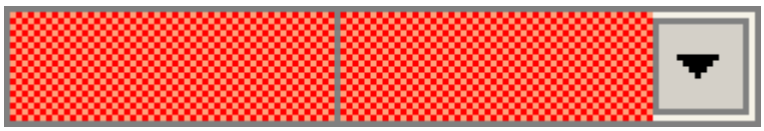
A cell with custom background is not filled with **BackColor** of the attached style, and iGrid fires the **CustomDrawCellBackground** event to give you the ability to draw the cell's background. The bounds provided by this event are the bounds of the cell except for grid lines.

The picture below shows three cells with a custom-drawn yellow hatched background (the real image was zoomed in):



iGrid does not draw the cell's icon and text for a cell with custom foreground. Instead, it fires the **CustomDrawCellForeground** event to give you the ability to draw the cell's foreground. The bounds provided by this event are the bounds of the cell except for the grid lines and the area occupied by the combo (for a combo box cell) and ellipsis button.

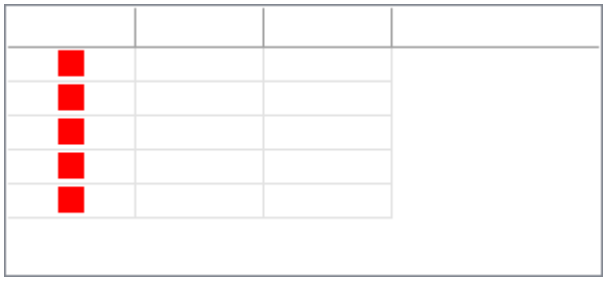
The picture below contains an example of cells with a hatched custom-drawn foreground:



You must also handle the **CustomDrawCellGetWidth** and **CustomDrawCellGetHeight** events for cells with custom foregrounds. These events are used to provide the width and height of the custom contents so that iGrid can automatically adjust column widths and row heights. This occurs when the **iGCol.AutoWidth** and **iGRow.AutoHeight** methods are invoked, or when the user double-clicks a column divider to automatically fit the width of the column.

The width of the custom contents must be passed in the **Width** property of the **CustomDrawCellGetWidth** event arguments.The height of the custom contents must be passed in the **Height** property of the **CustomDrawCellGetHeight** event arguments. These values exclude the internal cell indents and the thickness of the cell grid lines; it should represent solely the width and height of the custom contents.

As an example, let's build a grid with cells with a custom-drawn foreground. A 13x13 red rectangle will be rendered in the center of the cells in the first column. If the rectangle exceeds the available bounds, its size will be adjusted accordingly:



The grid setup code looks like this:

```
private void Form1_Load(object sender, System.EventArgs e)
{
  // Create columns and rows
  iGrid1.Cols.Count = 3;
  iGrid1.Rows.Count = 5;

  // Set custom draw flags
  iGrid1.Cols[0].CellStyle.CustomDrawFlags = iGCustomDrawFlags.Foreground;

  // Add event handlers
  iGrid1.CustomDrawCellForeground += iGrid1_CustomDrawCellForeground;
  iGrid1.CustomDrawCellGetWidth += iGrid1_CustomDrawCellGetWidth;
  iGrid1.CustomDrawCellGetHeight += iGrid1_CustomDrawCellGetHeight;
}
```

The custom drawing is implemented in the event handlers of the **CustomDrawCellForeground**, **CustomDrawCellGetWidth** and **CustomDrawCellGetHeight** events that are shown below. Pay

attention to the fact that we handle not only the **CustomDrawCellForeground** event, but also **CustomDrawCellGetWidth** and **CustomDrawCellGetHeight** because we provide the custom cell foreground:

```
private void iGrid1_CustomDrawCellForeground(
  object sender, iGCustomDrawCellEventArgs e)
{
  // Adjust the rectangle size taking into account
  // the bounds of the cell foreground area
  int myWidth = Math.Min(e.Bounds.Width, 13);
  int myHeight = Math.Min(e.Bounds.Height, 13);

  // Draw the rectangle
  e.Graphics.FillRectangle(
    Brushes.Red,
    e.Bounds.X + (e.Bounds.Width - myWidth)/2,
    e.Bounds.Y + (e.Bounds.Height - myHeight)/2,
    myWidth,
    myHeight);
}

private void iGrid1_CustomDrawCellGetWidth(
  object sender, iGCustomDrawCellGetWidthEventArgs e)
{
  // Set the desired width of the custom contents
  e.Width = 13;
}

private void iGrid1_CustomDrawCellGetHeight(
  object sender, iGCustomDrawCellGetHeightEventArgs e)
{
  // Set the desired height of the custom contents
  e.Height = 13;
}
```

The **CustomDrawCellBackground** and **CustomDrawCellForeground** events provide you with the information about the cell's state so you can implement different drawing in different states (normal, hot, pressed). Note that iGrid does not update a cell with custom-drawn contents when the mouse pointer enters/leaves it or the mouse button is pressed. If your custom-drawn cell must change its look in one of these events, you must invalidate the cell area and cause redrawing yourself.

6. SELECTION AND THE CURRENT CELL

6.1. Introduction to Selection in iGrid

One of the fundamental characteristics of cells that make up any grid control is the ability to be selected. Cell selection is a bit status indicating whether a cell is selected or not. As a rule, cell selection is used to mark cells for some operations — to point out a cell for editing, to calculate totals for a group of cells, to select a whole row for deletion, etc. Selected cells should look different compared to non-selected cells so that the user can distinguish them. iGrid uses the traditional way to indicate selected cells — rendering with special background and foreground colors.

Selection of cells in iGrid can be described in two aspects. The first aspect is how cells can be selected, i.e. the selection functionality itself. This includes the minimal selectable object (cell or row), whether several cells or rows can be selected simultaneously (multi-selection mode), the ability to make some cells unavailable for selection, and so on. The other aspect is how selected cells are highlighted on the screen to distinguish them from non-selected cells. The following two sections of this chapter describe these two aspects of cell selection in iGrid in greater detail.

6.2. Selection Concepts and Configuration Options

6.2.1. The Concept of the Current Cell, Row, Column

The current cell is the one that receives keyboard input when iGrid has input focus. This means that all keyboard input is sent to this cell. For example, if the user presses one of the keys that should start editing (for example, F2), the cell that is current is put into edit mode. Another good example are the cursor movement keys. When the user presses one of these keys, iGrid changes the current cell according to the pressed key: for example, the RIGHT ARROW key makes the next available cell in the row current.

As a rule, the current cell is automatically selected and additionally marked with a special focus rectangle drawn around the cell contents. The focus rectangle helps the user to see which cell is current if several cells can be selected simultaneously. According to the general concept of focus rectangle in the Windows OS, iGrid draws the focus rectangle around the current cell only when the grid control is focused. You can find out more about the iGrid focus rectangle from the Focus Rectangle topic in this chapter.

The current cell also defines the current row and the current column. The current row is the row containing the current cell, the current column is the column containing the current cell. If row selection mode is on (see the Cell-based and Row-based Selection topic for more information), iGrid automatically selects the current row and mark it with the focus rectangle instead of the current cell when iGrid is focused.

6.2.2. Cell-based and Row-based Selection

iGrid supports two selection modes that define the minimal portion of a selectable object. These are cell selection mode and row selection mode, or shortly 'cell mode' and 'row mode'. The default is cell mode in which the user can select individual cells. In row mode, the whole row becomes selected when the user clicks one of its cells. Row mode is activated by setting the Boolean **RowMode** property of iGrid to True:

```
iGrid1.RowMode = true;
```

When row mode is on, some keys that are generally used to change the current cell have other behavior that better corresponds to row mode:

- In cell selection mode, the LEFT ARROW/RIGHT ARROW keys change the current cell to the previous/next available cell. In row mode, these keys scroll the grid in the corresponding horizontal direction if it has the horizontal scroll bar.
- In cell selection mode, the HOME and END keys are used to select the first and last available cell in the current row. In row mode, these keys select the first and last available row in the grid.

The current cell is not visible in row mode by default — though it always exists. The Boolean **RowModeHasCurCell** property allows you to show the current cell in row mode. If you set this property to True, the current cell may be highlighted with its own background and foreground colors which are different from the row selection colors (see the **CurCellBackColor** and **CurCellForeColor** properties described in the Highlighting of the Current Cell topic). If the current cell is enabled in row mode, the ability to change the current cell inside the selected row using the traditional LEFT ARROW, RIGHT ARROW, HOME, and END keys is enabled too.

6.2.3. Selection Modes in iGrid

In iGrid, four selection modes are available. These modes can be used both in the regular cell-based selection mode and in row mode, where entire rows are selected instead of individual cells.

The behavior described in this topic applies to both of these cases. However, for simplicity, the explanations below refer to cell selection. When iGrid is in row mode, the same rules apply in the same way, except that rows are selected instead of cells.

The available selection modes are defined by the values of the **iGSelectionMode** enumeration:

- **None** — No cells can be selected.
- **One** — Only the current cell can be selected.
- **MultiSimple** — Multiple cells can be selected by clicking them.
- **MultiExtended** — Multiple cells can be selected by clicking them while holding down the CTRL key.

To choose a selection mode, set the **SelectionMode** property of iGrid to the desired **iGSelectionMode** value. The default is **One**.

Single selection mode

To select a cell, the user can click it or use the arrow keys to move to it.

Multi-simple selection mode

To select/deselect a cell, the user can click it, or move the current cell to it by using the arrow keys and then press the SPACE key.

To select a rectangular range of cells with the mouse, the user can first click the upper-left cell of the range, then hold down the SHIFT key and click the lower-right cell. If the **PressedMouseMoveMode** property is set to the **Selection** value, the user can also select a rectangular range by pointing to the first cell, pressing and holding the left mouse button, and then dragging the pointer to the last cell in the range.

To select a rectangular range of cells with the keyboard, the user can move the current cell to the first cell in the range, hold down the SHIFT key and move the current cell to the last cell in the group by using the arrow keys.

To deselect a rectangular range of cells with the mouse, the user can deselect the first cell in the range by clicking it, then hold down the SHIFT key and click the last cell in the range.

In this selection mode, all cells that are not directly involved in the selection or deselection operation keep their previous selected state, regardless of whether the CTRL key is pressed.

Multi-extended selection mode

To select a cell with the mouse, the user can click it. To keep the other selected cells selected, the user must hold down the CTRL key while clicking; otherwise, those cells are deselected.

To select or deselect a cell by using the keyboard, the user must hold down the CTRL key, move the current cell to the desired cell with the arrow keys, and then press the SPACE key. If the CTRL key is not held down when the current cell is moved with the arrow keys, the current cell becomes the only selected cell.

To select a rectangular range of cells with the mouse, the user can first click the upper-left cell of the range, then hold down the SHIFT key and click the lower-right cell. To preserve the selection state of the other cells, the user must also hold down the CTRL key while performing these steps. If the **PressedMouseMoveMode** property is set to the **Selection** value, the user can also select a rectangular range by pointing to the first cell, pressing and holding the left mouse button, and then dragging the pointer to the last cell in the range. In this case, the user must hold down the CTRL key as well to preserve the selection state of the other cells.

To select a rectangular range of cells by using the keyboard, the user must move the current cell to the first cell in the range, then hold down the SHIFT key and use the arrow keys to move the current cell to the last cell in the range. To preserve the selection state of the other cells, the user must hold down the CTRL key as well during these steps.

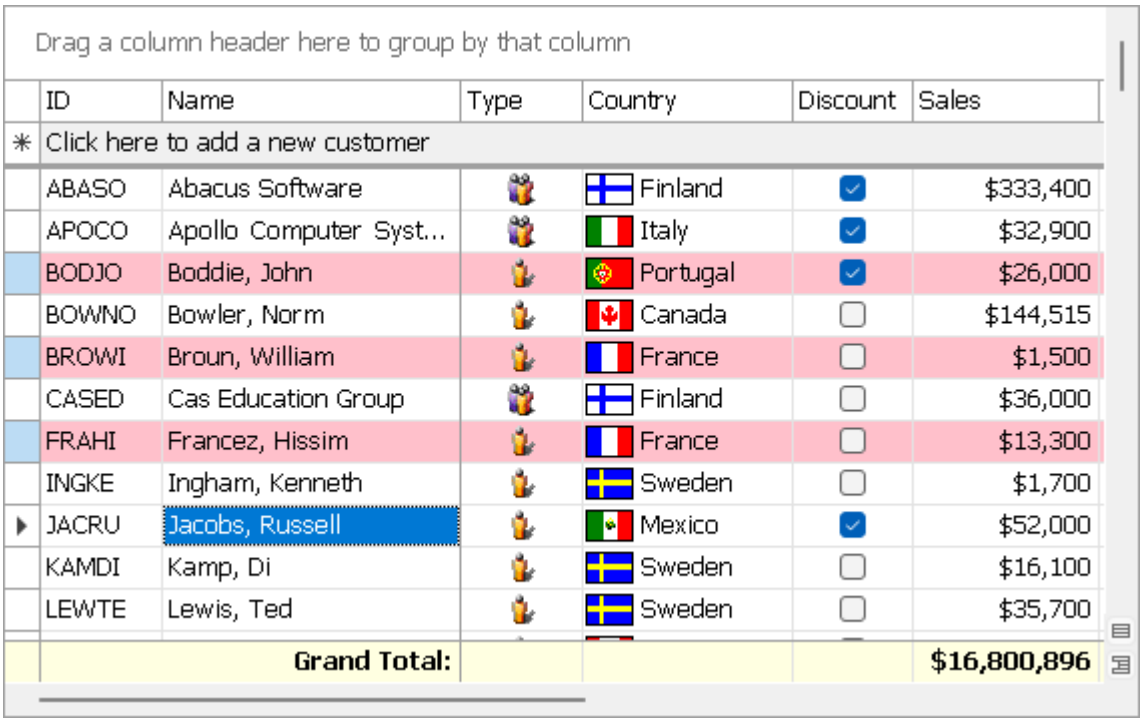
To deselect a cell with the mouse, the user must hold down the CTRL key and click the cell.

To deselect a rectangular range of cells with the mouse, the user must first hold down the CTRL key and click the first cell in the range to deselect it, then hold down the SHIFT key as well and click the last cell in the range.

6.2.4. Selecting Rows in Cell Selection Mode

iGrid provides a unique ability to select whole rows even if it works in cell selection mode. One of the scenarios in which this can be helpful is when you provide cell editing in cell selection mode and the ability to select several rows for other operations with rows — for example, for removal of selected rows. The user can still move the current cell without changing existing row selection.

The screenshot below demonstrates this functionality in action. You can see 3 selected rows marked with the pink color while iGrid still has the current cell available for editing:



This functionality is enabled with the **RowSelectionInCellMode** property of iGrid. The value of this property specifies how rows can be selected in cell mode:

- **None** — no rows can be selected (the default value).
- **SingleRow** — only one row can be selected.
- **MultipleRows** — several rows can be selected.

If row selection in cell mode is enabled, the row header area with row headers must be also visible because clicks on row headers are used to select rows in this special mode. The **Visible** property of the **RowHeader** object property of iGrid is used to show the row header area:

```
iGrid1.RowHeader.Visible = true;
```

If multiple rows can be selected in cell mode, the CTRL and SHIFT keys are used to select multiple rows while clicking their headers. These modifier keys work exactly like in the traditional multi-selection mode described in greater detail in the Selection Modes in iGrid topic.

To select a row, the user should click the header of the row. If the CTRL modifier key is pressed, the other rows will not change their selected state; otherwise this row will become the only selected row in the grid.

To deselect a row, the user should press and hold the CTRL modifier key and click the row's header. If this row is the only selected row, it is not obligatory to press the CTRL key.

To select/deselect a group of adjacent rows, the user should click the first row in the group, press and hold the SHIFT modifier key, and click the last row in the group. During these steps the user should hold the CTRL modifier key if he wants the other rows to preserve their selected state.

6.2.5. Enabling Selection of Invisible Cells

Columns and rows of iGrid can be made invisible for the user. If multi-selection mode is on, the user can select a range of cells using drag select or by clicking cells holding down the SHIFT modifier key (see the Selection Modes in iGrid topic for more information). By default the cells that are not currently visible on the screen are not included into selection during these operations.

The Boolean **SelectInvisibleCells** property of iGrid allows you to change this default behavior. Its default value equals False, which corresponds to the behavior described above. If you assign True to this property, iGrid will include invisible cells into range selection operations.

6.2.6. Protecting Cells and Rows from Being Selected

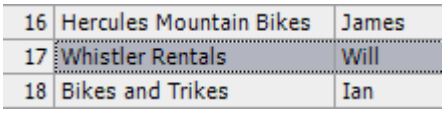
iGrid allows you to prevent cells from being selected by the user. To do it, use the following properties:

PROPERTY	CLASS	DESCRIPTION
Selectable	**iGCellStyle**	Specifies whether the cell can be selected through visual interface. If this property is set to False, the user cannot select this cell with the mouse or keyboard. When the user clicks such a cell, iGrid does not change the current cell and selection.
IncludeInSelect	**iGCol**	Specifies whether all the cells in the column can be selected. If this property is set to False, the user cannot select the cells with the mouse and keyboard. When the user clicks a cell in such a column, iGrid changes the selection as if the user clicked the cell in the same row but in the column which contained the current cell before that.

The **IncludeInSelect** property also affects the start and end columns of the selection in row mode. By default in row mode all the cells in a selected row are highlighted, but you can exclude some cells from the selection at left and right by setting the **IncludeInSelect** property to False. For example, if you do not want the cells of the first column to be highlighted, just set the **IncludeInSelect** property for that column to False:

```
iGrid1.Cols[0].IncludeInSelect = false
```

The result is on the picture below:



As with cells, you can make certain rows non-selectable. To do it, set their **Selectable** property to False. Note that when iGrid is in cell mode, the user can still select cells in non-selectable rows.

6.3. Highlighting the Current Cell, Selected Cells and Rows

6.3.1. Cell and Row Highlighting Properties

iGrid allows you to adjust the highlighting of the selected cells, rows, and the current cell. The table below lists the iGrid properties storing the corresponding color settings:

PROPERTY	DESCRIPTION
SelCellsBackColor	Determines the background color of the selected cells when the grid has input focus.
SelCellsBackColorNoFocus	Determines the background color of the selected cells when the grid does not have input focus.
SelCellsForeColor	Determines the color of the text in the selected cells when the grid has input focus.
SelCellsForeColorNoFocus	Determines the color of the text in the selected cells when the grid does not have input focus.
SelRowsBackColor	Determines the background color of the cells in the selected rows when the grid has input focus.
SelRowsBackColorNoFocus	Determines the background color of the cells in the selected rows when the grid does not have input focus.
SelRowsForeColor	Determines the color of the cell text in the selected rows when the grid has input focus.
SelRowsForeColorNoFocus	Determines the color of the cell text in the selected rows when the grid does not have input focus.
CurCellBackColor	Determines the background color of the current cell when the grid has input focus.
CurCellBackColorNoFocus	Determines the background color of the current cell when the grid does not have input focus.

PROPERTY	DESCRIPTION
CurCellForeColor	Determines the color of the text in the current cell when the grid has input focus.
CurCellForeColorNoFocus	Determines the color of the text in the current cell when the grid does not have input focus.

Selected cells and selected rows can be highlighted with different colors. When a cell is selected, it is highlighted with the colors specified in the **SelCellsBackColor**, **SelCellsBackColorNoFocus**, **SelCellsForeColor**, and **SelCellsForeColorNoFocus** properties. When a row is selected, it is highlighted with the colors specified in the **SelRowsBackColor**, **SelRowsBackColorNoFocus**, **SelRowsForeColor**, and **SelRowsForeColorNoFocus** properties. If a **SelRows*** property is not specified (contains **Color**.**Empty**), the color from the corresponding **SelCells*** property is used. By default, all these **SelRows*** contain **Color**.**Empty**, and iGrid uses the same highlight colors for select rows and selected cells defined in the **SelCells*** properties.

The same system is used to highlight the current cell. By default, the current cell is also highlighted with the colors of selected cells specified in the **SelCells*** properties. You can make the current cell highlighted with special colors by using the **CurCellBackColor**, **CurCellBackColorNoFocus**, **CurCellForeColor**, and **CurCellForeColorNoFocus** properties.

iGrid implements two more properties allowing you to specify whether selected cells or their parts should be highlighted:

PROPERTY	DESCRIPTION
HighlightSelCellItems	Determines which cell items, controls or images, should be highlighted in selected cells.
HighlightSelCells	Gets or sets a value indicating whether selected cells should use the highlight effect.

The remaining topics in this section describe how the values of these properties are used by iGrid.

6.3.2. Highlighting of Selected Cells

The selected cell background is highlighted with the colors specified by the **SelCellsBackColor** and **SelCellsBackColorNoFocus** properties. The former applies when iGrid has input focus, and the latter when it does not.

The highlighting of selected cells can be performed in two ways: with a semi-transparent background color or an opaque background color. If the selected cell background color is set to a semi-transparent color, the cell background will be a blend of the cell selection background color, row selection background color, current cell background color, and cell original background color. The full algorithm of highlighting cell background can be found in the Putting All Related to Cell Highlighting Together topic below.

The text in selected cells is highlighted with the colors specified by the **SelCellsForeColor** and **SelCellsForeColorNoFocus** properties. The former applies when iGrid has focus, and the latter when it does not.

6.3.3. Highlighting of Selected Rows

Selected rows are highlighted with the colors specified by the **SelRowsBackColor**, **SelRowsBackColorNoFocus**, **SelRowsForeColor**, and **SelRowsForeColorNoFocus** properties.

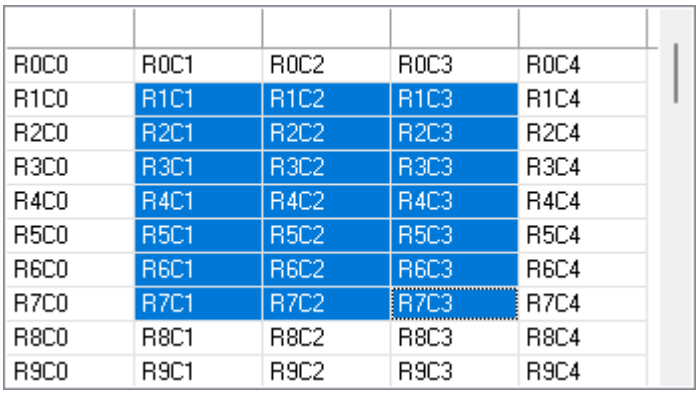
You can highlight selected rows with a semi-transparent background color. In this case the background of a cell located in a selected row will be a blend of the cell selection background color, row selection background color, current cell background color, and cell original background color. The full algorithm of highlighting cell background can be found in the Putting All Related to Cell Highlighting Together topic below.

If you set a selected row color to **Color**.**Empty**, the selected rows will be highlighted with the same color as selected cells. It is the default behavior.

6.3.4. Highlighting of the Current Cell

By default the current cell is highlighted with the colors of selected cells, i.e. the colors specified by the **SelCellsBackColor** and **SelCellsForeColor** properties of iGrid. You can use special colors to highlight the current cell that may differ from the standard selection colors. This is done with the help of the **CurCellBackColor** and **CurCellForeColor** properties of iGrid. The default value for both properties is **Color**.**Empty**, which means that the corresponding color is not set and iGrid should highlight the current cell using the selection colors

This feature can be used to highlight the current cell among all selected cells if multi-selection mode is on. Look at the following grid in which multi-selection mode is enabled and a range of cells is selected:

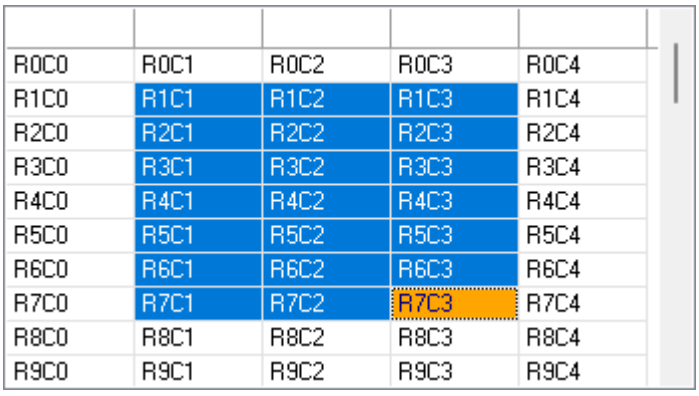


R0C0	R0C1	R0C2	R0C3	R0C4
R1C0	R1C1	R1C2	R1C3	R1C4
R2C0	R2C1	R2C2	R2C3	R2C4
R3C0	R3C1	R3C2	R3C3	R3C4
R4C0	R4C1	R4C2	R4C3	R4C4
R5C0	R5C1	R5C2	R5C3	R5C4
R6C0	R6C1	R6C2	R6C3	R6C4
R7C0	R7C1	R7C2	R7C3	R7C4
R8C0	R8C1	R8C2	R8C3	R8C4
R9C0	R9C1	R9C2	R9C3	R9C4

The R7C3 cell, which is the current cell, is hardly noticeable. We can significantly improve the situation with the following settings:

```
iGrid1.CurCellBackColor = Color.Orange;
iGrid1.CurCellForeColor = Color.DarkBlue;
```

The result is on the screenshot below:



R0C0	R0C1	R0C2	R0C3	R0C4
R1C0	R1C1	R1C2	R1C3	R1C4
R2C0	R2C1	R2C2	R2C3	R2C4
R3C0	R3C1	R3C2	R3C3	R3C4
R4C0	R4C1	R4C2	R4C3	R4C4
R5C0	R5C1	R5C2	R5C3	R5C4
R6C0	R6C1	R6C2	R6C3	R6C4
R7C0	R7C1	R7C2	R7C3	R7C4
R8C0	R8C1	R8C2	R8C3	R8C4
R9C0	R9C1	R9C2	R9C3	R9C4

Another example when this feature is used is when the current cell is shown in row selection mode. This feature is enabled when you set the **RowModeHasCurCell** property of iGrid to True. If you used iGrid with the default color settings, you would not see the current cell because it is highlighted with the selection colors used to highlight all cells in the current row. The example below shows how to make the current cell noticeable in this situation:

```
iGrid1.RowMode = true;
iGrid1.RowModeHasCurCell = true;

iGrid1.CurCellBackColor = Color.Pink;
iGrid1.CurCellForeColor = Color.Black;
```

The current cell's background color can also be set to a semi-transparent color. In this case the background of the current cell will be a blend of the cell selection background color, row selection background color, current cell background color, and cell original background color. The full algorithm of highlighting cell background can be found in the Putting All Related to Cell Highlighting Together topic below.

6.3.5. Putting All Related to Cell Highlighting Together

The full step-by-step procedure explaining how iGrid uses the selection-related properties while drawing a cell is described below. The concept of cell items is used at that. Cell items in this context are non-text cell elements, such as cell image and cell controls (combo button, check box and the like). The **HighlightSelCellItems** property of iGrid specifies which cell items should be highlighted when the cell or row it is located in is selected (regardless of whether selection colors are transparent).

1. The grid's background is drawn.
2. The own background of the cell is drawn. This is either a rectangle filled with the background color specified in the **BackColor** property of the cell (which can be redefined with the **CellDynamicFormatting** event of iGrid) or a custom-drawn background (see the **CustomDrawCellBackground** event of iGrid).
3. If the cell belongs to a selected row, and **SelRowsBackColor** is opaque, the cell is filled with **SelRowsBackColor**. If **SelRowsBackColor** is not opaque and the cell items should not be highlighted, they are filled with **SelRowsBackColor**.
4. If the cell itself is selected, and **SelCellsBackColor** is opaque, the cell is filled with **SelCellsBackColor**. If **SelCellsBackColor** is semi-transparent and the cell items should not be highlighted, they are filled with **SelCellsBackColor**.
5. If the cell is the current cell, **CurCellBackColor** is opaque and not empty, the cell is filled with **CurCellBackColor**. If **CurCellBackColor** is semi-transparent and the cell items should not be highlighted, they are filled with **CurCellBackColor**.
6. If cell items (cell image and cell controls) are specified, they are drawn.
7. If the cell belongs to a selected row and the cell items should be highlighted, they are filled with **SelRowsBackColor** if it is semi-transparent or a semi-transparent equivalent derived from **SelRowsBackColor** if it is opaque.
8. If the cell itself is selected and the cell items should be highlighted, they are filled with **SelCellsBackColor** if it is semi-transparent or a semi-transparent equivalent derived from **SelCellsBackColor** if it is opaque.
9. If the cell is the current cell and the cell items should be highlighted, they are filled with **CurCellBackColor** if it is semi-transparent or a semi-transparent equivalent derived from **CurCellBackColor** if it is opaque.
10. The cell text is drawn (if any).

6.4. Working with Selection in Code

6.4.1. Working with the Current Cell, Row, Column from Code

To determine the current cell from code, use the **CurCell** property of iGrid. It returns an instance of the **iGCell** class that represents the current cell. Its properties **RowIndex** and **ColIndex** can be used to know the row/column the current cell is placed in.

You can also use this property to make a cell current. To do that, assign an **iGCell** object that represents the new current cell to this property, for instance:

```
iGrid1.CurCell = iGrid1.Cells[2, 1];
```

Note that the value returned by the **CurCell** property may be null (Nothing in VB), which indicates that iGrid does not have the current cell. If you want to do something with the current cell in your code, first always test whether the value returned by the **CurCell** property is not null.

This property can be used to make no cell current:

```
iGrid1.CurCell = null;
```

The overloaded versions of the **SetCurCell** method allow you to specify the current cell using string column and row keys. The following example shows how to make the cell with the row index 3 and column key "Country" current:

```
iGrid1.SetCurCell(2, "Country");
```

When you change the current cell using one of the methods described above, iGrid automatically scrolls its contents so that the current cell becomes visible in the viewport. Note that screen updates must be enabled in iGrid to show the current cell in the viewport automatically. In other words, if you change the current cell from code between the **BeginUpdate** and **EndUpdate** method calls, it is not guaranteed that the new current cell will appear in the visible area of the grid.

The current row can be accessed with the **CurRow** property. To set the current row, use one of the overloaded versions of the **SetCurRow** method. The following code shows how to change the current row:

```
iGrid1.SetCurRow(2);
```

iGrid implements similar members to work with the current column from code — the **CurCol** property and the **SetCurCol** method.

6.4.2. Working with the Selected Cells and Rows from Code

Enumerating the selected cells and rows

The **SelectedCells** property of iGrid returns the collection of the currently selected cells. This collection is read-only and allows you to enumerate all the selected cells.

The **SelectedRows** property of iGrid returns the collection of the currently selected rows. This collection is read-only and allows you to enumerate all the selected rows.

Selecting cells

If you need to select a cell from code, set the **Selected** property of the corresponding **iGCell** object to True.

You can use the **iGCell**.**Selected** property to select a range of cells from code:

```
private void SelectRangeOfCells(
  int startRowIndex, int startColIndex,
  int endRowIndex, int endColIndex)
{
  iGrid1.BeginUpdate();

  for(int iRow = startRowIndex; iRow <= endRowIndex; iRow++)
    for(int iCol = startColIndex; iCol <= endColIndex; iCol++)
      iGrid1.Cells[iRow, iCol].Selected = true;

  iGrid1.EndUpdate();
}
```

Pay attention to the **BeginUpdate**/**EndUpdate** method calls. If we did not wrap the loops with these method calls, iGrid would redraw its contents on the screen after every change of the **Selected** property, and our code would work much slower.

iGrid implements the **SelectCellRange** method that performs the same task. In the vast majority of cases the above code snippet can be replaced with the following statement:

```
iGrid1.SelectCellRange(
  startRowIndex, startColIndex, endRowIndex, endColIndex);
```

The **SelectCellRange** method performs the same task up to 40% faster due to some optimizations. First, it does not raise the **SelectionChanged** event after changing the selection status of every processed cell — the **SelectionChanged** event is raised only once at the end of the whole operation. Second, the intermediary **iGCell** object is not created to access the **Selected** property of every cell.

If you want to select all the cells in iGrid, invoke the **PerformAction** method and pass the **iGActions**.**SelectAllCells** value to it:

```
iGrid1.PerformAction(iGActions.SelectAllCells);
```

Selecting rows

If you want to select a row from code, set the **Selected** property of the corresponding **iGRow** object to True.

The following example shows how to select a range of rows from code:

```
private void SelectRangeOfRows(int startRowIndex, int endRowIndex)
{
  iGrid1.BeginUpdate();

  for(int iRow = startRowIndex; iRow <= endRowIndex; iRow++)
    iGrid1.Rows[iRow].Selected = true;

  iGrid1.EndUpdate();
}
```

Like in the case of cells, you can use the equivalent **SelectRowRange** method if you are not interested in processing the **SelectionChanged** event.

If you want to select all the rows in iGrid, invoke the **PerformAction** method and pass the **iGActions**.**SelectAllRows** value to it.

Deselecting cells and rows

The **SelectedCells** and **SelectedRows** collections are read-only. This means that you cannot change the selection in iGrid by adding/removing items to/from these collections.

If you need to deselect an individual cell, set the **Selected** property of the corresponding **iGCell** object to False. The same concerns a row, but in this case you use the **Selected** property of the corresponding row object (**iGRow**).

To deselect a range of cells or rows, use the **DeselectCellRange** or **DeselectRowRange** methods of iGrid respectively.

If you want to deselect all the selected cells, you can use the **PerformAction** method for that — simply issue it with the **DeselectAllCells** parameter:

```
iGrid1.PerformAction(iGActions.DeselectAllCells);
```

To deselect all rows in one statement, use the **DeselectAllRows** action:

```
iGrid1.PerformAction(iGActions.DeselectAllRows);
```

6.4.3. iGrid Events Related to Selection

The Current Cell, Column, and Row

When the user is about to change the current cell, iGrid raises the **CurCellChangeRequest** event. This event occurs before the current cell may be changed and allows you to approve or reject this change if required with the **DoDefault** field of the event arguments object. If the current cell has been changed, iGrid raises the **CurCellChanged** event. The fields of the event arguments object of this event allow you to know the previous current cell and the new current cell.

One important note regarding the **CurCellChanged** event. iGrid does not raise this event in cases when the developer adds or removes rows/columns before the current cell. This is explained by the fact that in these situations the current cell is actually not changed though the row and/or column index of the current cell may be changed.

To control and monitor changes of the current column and row, iGrid provides a similar set of events for these objects: **CurColChangeRequest**, **CurColChanged**, **CurRowChangeRequest**, and **CurRowChanged**.

Cell or Row Selection

To monitor changes of selection status of cells and rows, use the **SelectionChanged** event. This event is raised both when the user has changed the selection through visual interface and you have changed the selection from code. Note that selection change is not the same as the current cell change in the general case. If iGrid works in single-selection mode, the selection is changed together with the current cell. But if multi-selection is on, the user can change selection without changing the current cell — for example, if the current cell is clicked while holding down the CTRL key to toggle the selection of the current cell.

Drag Select Operation

If multiple selection is enabled and the **PressedMouseMoveMode** property of iGrid is set to the **Selection** value, the user can select a range of cells or rows in row mode by moving the mouse pointer while holding down the mouse button. This operation is called 'drag select'. To monitor the selection change during drag select, use the **DragSelectStart**, **DragSelectRangeChange**, and **DragSelectEnd** events.

The **DragSelectStart** event occurs when the user starts a drag-select operation. The **DragSelectRangeChange** event occurs whenever the drag-select range changes while the mouse

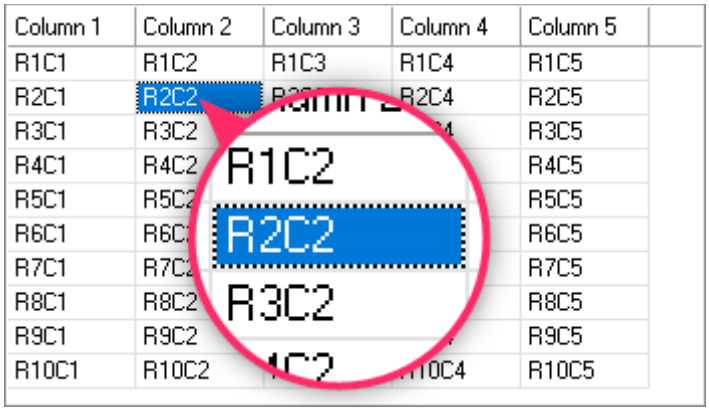
pointer is being moved with the mouse button still pressed. The **DragSelectEnd** event occurs when the user releases the mouse button and the drag-select operation ends.

Note that during drag select, iGrid does not change the actual selection immediately. Until the mouse button is released, it only marks the range that will be selected by using a special semi-transparent selection color. Each time this marked range changes during the operation, iGrid raises the **DragSelectRangeChange** event. The actual selection is updated only when the drag-select operation is completed.

When the drag-select operation ends, iGrid raises both the **DragSelectEnd** event and the general **SelectionChanged** event, which informs you that the actual selection has changed.

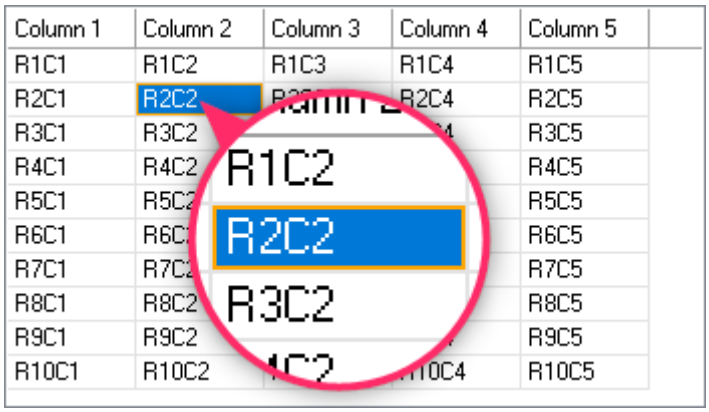
6.5. Focus Rectangle

By default iGrid marks the current cell accepting the keyboard input with a dotted focus rectangle:



The focus rectangle can be turned off with the Boolean **FocusRect** property of iGrid. Set it to False to disable drawing of the focus rectangle.

The values of the **FocusRectColor1** and **FocusRectColor2** properties of iGrid specify the colors used to draw the focus rectangle. You can use them to change the alternating dot colors of the focus rectangle. You can even make a solid focus rectangle if you set both properties to the same value, for example:



The focus rectangle above was created with the following code:

```
iGrid1.FocusRectColor1 = Color.Orange;
iGrid1.FocusRectColor2 = Color.Orange;
```

Note that the focus rectangle is drawn in iGrid only when the control has input focus.

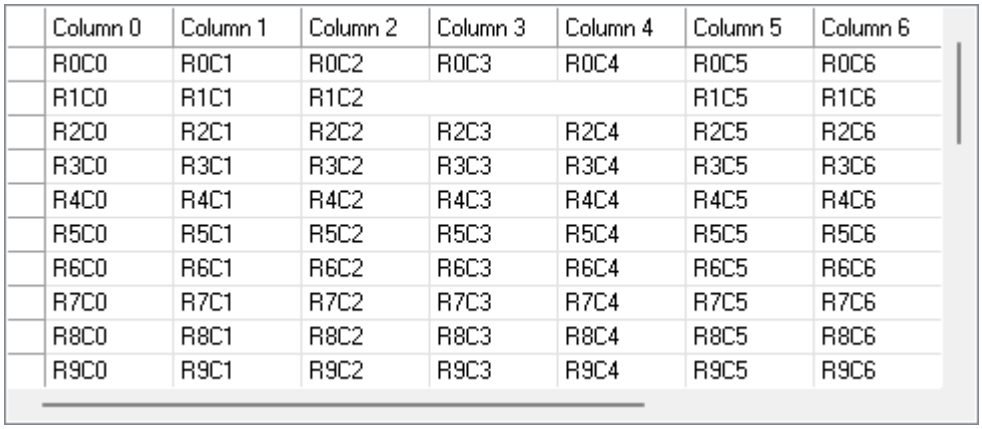
7. CELL MERGING

7.1. Basics of Cell Merging

iGrid allows you to merge its cells. A cell can be merged with adjacent cells at the right and adjacent cells at the bottom. The **SpanCols** and **SpanRows** properties of the **iGCell** object are used to specify the number of columns and rows respectively a cell should span. The default values for these properties are 1, which means that every cell of iGrid is not merged with other cells. Setting these properties to a positive value greater than 1 leads to merging of cells in the corresponding direction. For instance, if we want to merge the third cell in the second row with the cells in the adjacent 2 columns at the right, we can use the following code:

```
iGrid1.Cells[1, 2].SpanCols = 3;
```

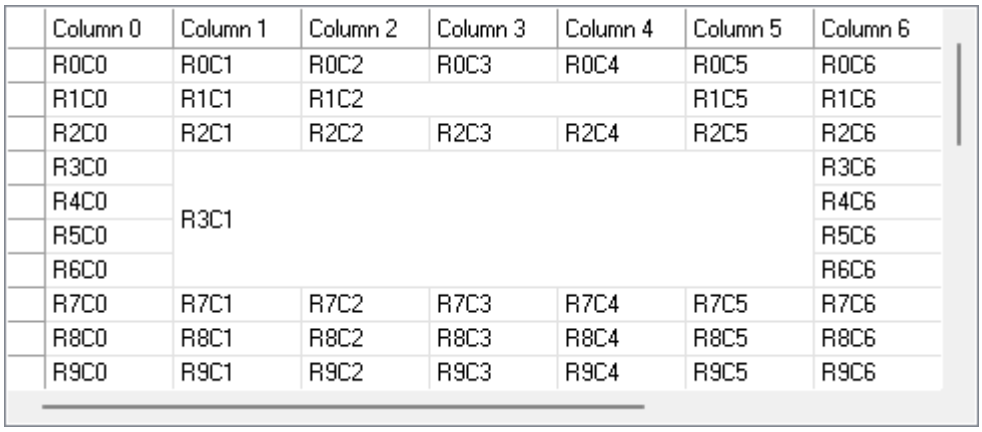
The result is depicted on the following screenshot:



	Column 0	Column 1	Column 2	Column 3	Column 4	Column 5	Column 6
	R0C0	R0C1	R0C2	R0C3	R0C4	R0C5	R0C6
	R1C0	R1C1	R1C2			R1C5	R1C6
	R2C0	R2C1	R2C2	R2C3	R2C4	R2C5	R2C6
	R3C0	R3C1	R3C2	R3C3	R3C4	R3C5	R3C6
	R4C0	R4C1	R4C2	R4C3	R4C4	R4C5	R4C6
	R5C0	R5C1	R5C2	R5C3	R5C4	R5C5	R5C6
	R6C0	R6C1	R6C2	R6C3	R6C4	R6C5	R6C6
	R7C0	R7C1	R7C2	R7C3	R7C4	R7C5	R7C6
	R8C0	R8C1	R8C2	R8C3	R8C4	R8C5	R8C6
	R9C0	R9C1	R9C2	R9C3	R9C4	R9C5	R9C6

We can merge cells in columns and rows simultaneously, for example:

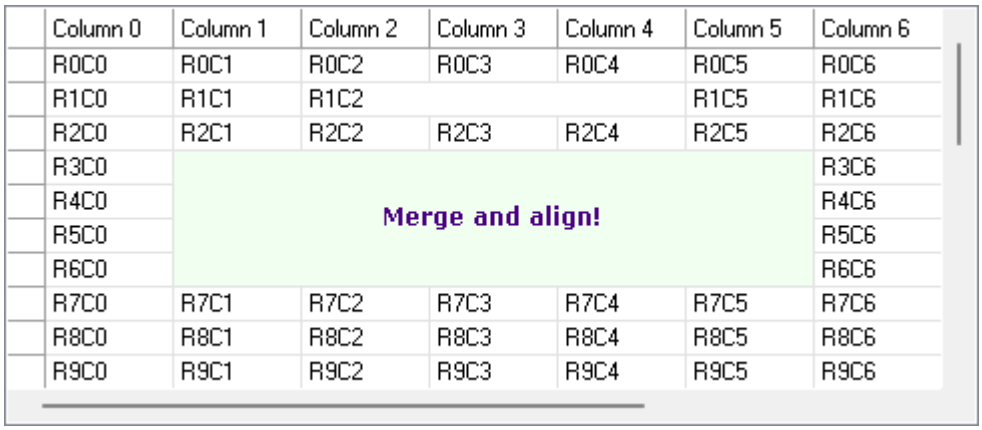
```
iGCell myCell = iGrid1.Cells[3, 1];
myCell.SpanCols = 5;
myCell.SpanRows = 4;
```



	Column 0	Column 1	Column 2	Column 3	Column 4	Column 5	Column 6
	R0C0	R0C1	R0C2	R0C3	R0C4	R0C5	R0C6
	R1C0	R1C1	R1C2			R1C5	R1C6
	R2C0	R2C1	R2C2	R2C3	R2C4	R2C5	R2C6
	R3C0	R3C1					R3C6
	R4C0						R4C6
	R5C0						R5C6
	R6C0						R6C6
	R7C0	R7C1	R7C2	R7C3	R7C4	R7C5	R7C6
	R8C0	R8C1	R8C2	R8C3	R8C4	R8C5	R8C6
	R9C0	R9C1	R9C2	R9C3	R9C4	R9C5	R9C6

The cell for which we increased the **SpanCols** and/or **SpanRows** properties becomes the "root" of the whole merged cell and determines its look and behavior. You can use all available rich set of cell properties available for non-merged cells to format a merged cell. For instance, we can add the following code to the code snippet above to format our second merged cell in the grid:

```
myCell.Value = "Merge and align!";
myCell.TextAlign = iGContentAlignment.MiddleCenter;
myCell.Font = new Font("Verdana", 9, FontStyle.Bold);
myCell.BackColor = Color.Honeydew;
myCell.ForeColor = Color.Indigo;
```



A merged cell looks like its expanded root cell covering other normal cells, but these normal cells are not removed from iGrid and they still exists. You can access and change their properties — though you will not see the result of your actions. If you want to know the root of the merged cell covering a normal cell, you can always access it **SpanRoot** property.

You can retrieve the collection of currently merged cells with the **MergedCells** property of iGrid. This collection is ordered by row and column indexes of merged cell roots and can be used in for-each loops. The following example demonstrates how to print the row and column indexes of merged cells in the Output window in Visual Studio:

```
foreach (iGCell item in iGrid1.MergedCells)
{
   Debug.WriteLine($"({item.RowIndex}, {item.ColIndex})");
}
```

Some final notes regarding merged cells:

1. When you specify the number of columns to span in the **SpanCols** property of a cell, iGrid merges cells using the current visible column order.
2. If iGrid works in right-to-left mode, the **SpanCols** cell property specifies the number of columns to span to the left and the root of a merged cell is located at the top-right corner of the merged cell.
3. The cell pattern class **iGCellPattern** also provides you with the **SpanCols** and **SpanRows** properties that can be used to define patterns of merged cells.

7.2. Methods for Merging/Unmerging Cells

Merging cells in columns

Many grids in real-world applications display tables from relational databases. As a rule, those tables may contain a column or several columns containing repeated values if the table rows contain data about the same object. When you display such a table in a grid control, you may want to show adjacent cells with the same values in those columns like one merged cell for better visual perception. iGrid allows you to do that easily with the **MergeCellsInCols** method.

Let's consider the following sample grid:

Customer ↑1	Order ↑2	Item	Price
Customer 1	Order 1	Item 3-823	27.34
Customer 1	Order 1	Item 3-688	66.70
Customer 1	Order 1	Item 6-389	52.24
Customer 1	Order 2	Item 6-718	29.33
Customer 1	Order 2	Item 4-225	44.18
Customer 1	Order 2	Item 1-475	60.53
Customer 1	Order 3	Item 3-307	40.39
Customer 1	Order 3	Item 3-440	46.53
Customer 2	Order 1	Item 1-913	75.69
Customer 2	Order 1	Item 1-142	95.43
Customer 2	Order 1	Item 6-837	99.32
Customer 2	Order 1	Item 1-620	79.93
Customer 2	Order 2	Item 3-676	77.77
Customer 2	Order 2	Item 5-709	59.62
Customer 2	Order 2	Item 6-213	25.11
Customer 2	Order 2	Item 2-573	85.11
Customer 2	Order 3	Item 6-846	10.11
Customer 3	Order 1	Item 4-361	46.19
Customer 3	Order 1	Item 1-353	64.83

Its first two columns contain repeated values telling us which rows belong to the same customer and placed orders. If we want to group rows belonging to one customer with merged cells, we can use the following call:

```
iGrid1.MergeCellsInCols(0);
```

The result is on the picture below:

Customer ↑1	Order ↑2	Item	Price
Customer 1	Order 1	Item 3-823	27.34
	Order 1	Item 3-688	66.70
	Order 1	Item 6-389	52.24
	Order 2	Item 6-718	29.33
	Order 2	Item 4-225	44.18
	Order 2	Item 1-475	60.53
	Order 3	Item 3-307	40.39
	Order 3	Item 3-440	46.53
Customer 2	Order 1	Item 1-913	75.69
	Order 1	Item 1-142	95.43
	Order 1	Item 6-837	99.32
	Order 1	Item 1-620	79.93
	Order 2	Item 3-676	77.77
	Order 2	Item 5-709	59.62
	Order 2	Item 6-213	25.11
	Order 2	Item 2-573	85.11
	Order 3	Item 6-846	10.11
	Order 1	Item 4-361	46.19
	Order 1	Item 1-353	64.83

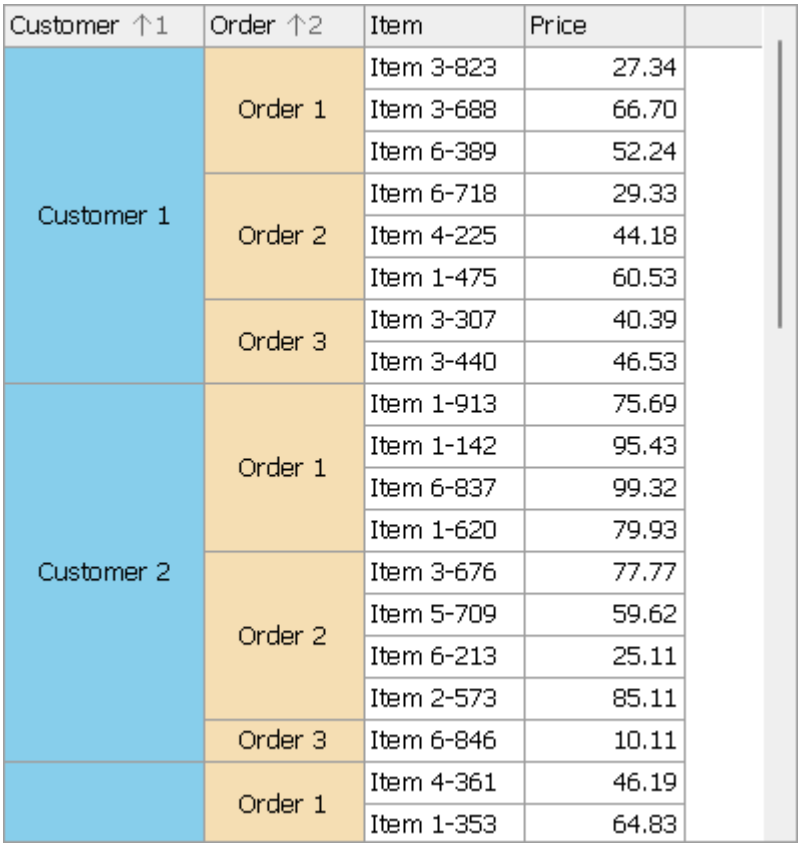
As the method name "MergeCellsInCols" implies, the method can be used to merge cells in more than one column. There are ten overloaded versions of the method allowing us to specify a column or columns to merge cells in using column numeric indexes or column string keys. In our sample grid above we defined string keys for the Customer and Order columns, and we can choose the most convenient overloaded version of the **MergeCellsInCols** method to merge cells in these two columns:

```
iGrid1.MergeCellsInCols(new string[] { "Customer", "Order" });
```

Customer ↑1	Order ↑2	Item	Price
Customer 1	Order 1	Item 3-823	27.34
		Item 3-688	66.70
		Item 6-389	52.24
	Order 2	Item 6-718	29.33
		Item 4-225	44.18
		Item 1-475	60.53
	Order 3	Item 3-307	40.39
		Item 3-440	46.53
Customer 2	Order 1	Item 1-913	75.69
		Item 1-142	95.43
		Item 6-837	99.32
		Item 1-620	79.93
	Order 2	Item 3-676	77.77
		Item 5-709	59.62
		Item 6-213	25.11
		Item 2-573	85.11
	Order 3	Item 6-846	10.11
	Order 1	Item 4-361	46.19
		Item 1-353	64.83

Note that if we specify several columns in a **MergeCellsInCols** call, cells with the same value in the second specified column are merged into one only if they belong to the same row range defined by the same cell value from the first specified column, cells with the same value in the third specified column are merged into one only if they belong to the same row range defined by the same cell values from the first and second specified columns, and so on. Thus, calling this method for several columns is not the same as calling the **MergeCellsInCols** method for every column separately.

The other overloaded versions of the **MergeCellsInCols** method allow us to apply cell styles to columns we merge cells in to produce a more vivid picture like this:



Customer ↑1	Order ↑2	Item	Price
Customer 1	Order 1	Item 3-823	27.34
		Item 3-688	66.70
		Item 6-389	52.24
	Order 2	Item 6-718	29.33
		Item 4-225	44.18
		Item 1-475	60.53
	Order 3	Item 3-307	40.39
		Item 3-440	46.53
Customer 2	Order 1	Item 1-913	75.69
		Item 1-142	95.43
		Item 6-837	99.32
		Item 1-620	79.93
	Order 2	Item 3-676	77.77
		Item 5-709	59.62
		Item 6-213	25.11
		Item 2-573	85.11
	Order 3	Item 6-846	10.11
	Order 1	Item 4-361	46.19
		Item 1-353	64.83

We got this picture after executing the following code snippet:

```
iGCellStyle myMergedCellStyle1 = iGrid1.Cols["Customer"].CellStyle.Clone();
myMergedCellStyle1.TextAlign = iGContentAlignment.MiddleCenter;
myMergedCellStyle1.BackColor = Color.SkyBlue;

iGCellStyle myMergedCellStyle2 = myMergedCellStyle1.Clone();
myMergedCellStyle2.BackColor = Color.Wheat;

iGrid1.MergeCellsInCols(
    new string[] { "Customer", "Order" },
    new iGCellStyle[] { myMergedCellStyle1, myMergedCellStyle2 });
```

For real data, you may get merged cells with heights that exceed the grid viewport height. Chances are very high that your users will see blank parts of those cells after scrolling, but you can force iGrid to draw the contents of merged cells in the viewport. For more information, read the Forcing iGrid to Draw Cell Contents in Viewport topic.

Unmerging cells

If you merged cells in a column or columns with one of the overloaded versions of the **MergeCellsInCols** method, you can revert your changes with the **UnmergeCellsInCols** method. If you applied custom styles to merged cells, you can optionally remove them too.

iGrid also provides you with the **UnmergeAllCells** method that can be used to remove merging from all merged cells in one call.

7.3. Merged Cells in Auto-sizing Operations

iGrid provides three methods for automatically adjusting column widths and row heights when it contains merged cells. You can separately specify the method used by iGrid for columns and rows

with the **AutoWidthColSpanHandling** and **AutoHeightRowSpanHandling** properties of iGrid. Both properties accept one of the values from the **iGAutoSizeSpanHandling** enumeration:

```
public enum iGAutoSizeSpanHandling
{
  Ignore = 0,             // Ignore merged cells during auto-sizing
  SizeRootUsingSpan = 1,  // Size root using info from entire span
  SizeRootOnly = 2        // Size root using only root's own info
}
```

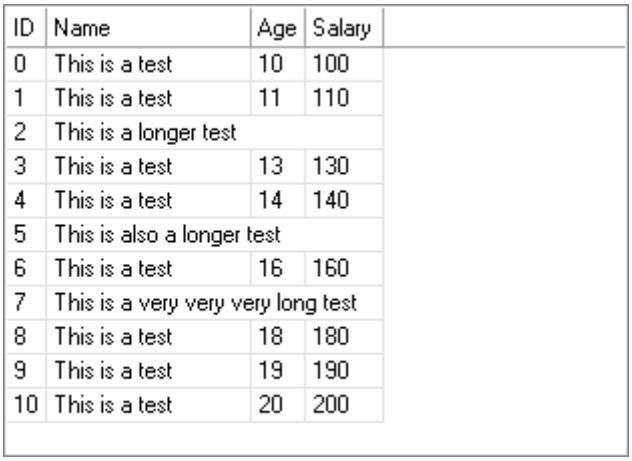
To demonstrate how these methods work, let's consider the following grid with several merged cells spanning over columns:

ID	Name	Age	Salary
0	This is a test	10	100
1	This is a test	11	110
2	This is a longer test		
3	This is a test	13	130
4	This is a test	14	140
5	This is also a longer test		
6	This is a test	16	160
7	This is a very very very long...		
8	This is a test	18	180
9	This is a test	19	190
10	This is a test	20	200

The default value for the **AutoWidthColSpanHandling** property is **Ignore**. in this case, if the user double-clicks the divider of the Name column or the developer calls the **iGCol**.**AutoWidth** method to automatically adjust the column's width, iGrid adjusts the width of the column considering only non-merged cells in the column:

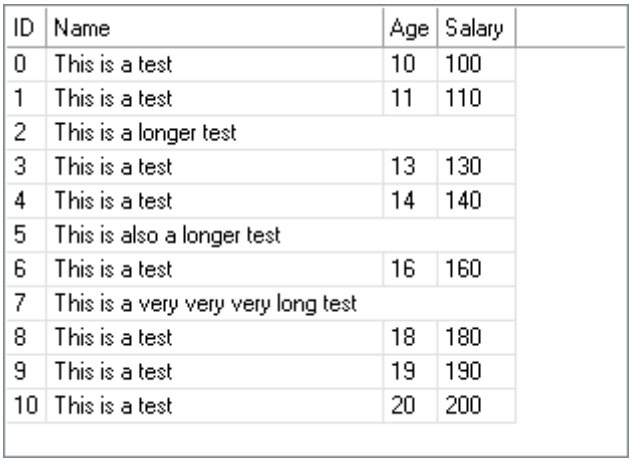
ID	Name	Age	Salary
0	This is a test	10	100
1	This is a test	11	110
2	This is a longer test		
3	This is a test	13	130
4	This is a test	14	140
5	This is also a longer test		
6	This is a test	16	160
7	This is a very very very l...		
8	This is a test	18	180
9	This is a test	19	190
10	This is a test	20	200

If you set the **AutoWidthColSpanHandling** property to **SizeRootUsingSpan**, iGrid sets the width of the Name column to the minimal value at which the texts of the merged cells become fully visible:



ID	Name	Age	Salary
0	This is a test	10	100
1	This is a test	11	110
2	This is a longer test		
3	This is a test	13	130
4	This is a test	14	140
5	This is also a longer test		
6	This is a test	16	160
7	This is a very very very long test		
8	This is a test	18	180
9	This is a test	19	190
10	This is a test	20	200

The last auto-sizing method enabled with the **SizeRootOnly** member causes iGrid to enlarge the Name column so that its width is enough to display the texts of merged cells without clipping:



ID	Name	Age	Salary
0	This is a test	10	100
1	This is a test	11	110
2	This is a longer test		
3	This is a test	13	130
4	This is a test	14	140
5	This is also a longer test		
6	This is a test	16	160
7	This is a very very very long test		
8	This is a test	18	180
9	This is a test	19	190
10	This is a test	20	200

The **Ignore** mode is selected by default because it mirrors behavior in widely used applications such as Microsoft Excel, aligning with user expectations. The **SizeRootUsingSpan** mode is the smartest mode that allows you to set the widths of all grid columns to some optimal values to display the texts of all cells without clipping. You can do this in one call to the **iGrid**.**Cols**.**AutoWidth** method because it also respects the value of the **AutoWidthColSpanHandling** property of iGrid.

Everything written above also applies to grid rows, with the difference that the described methods work in the "vertical direction".

7.4. Restrictions in iGrid with Merged Cells

Main restrictions related to merged cells

If iGrid contains merged cells, some operations you could do from code or interactively become unavailable. These restrictions are caused by the following two basic principles of merged cells:

1. A merged cell cannot be separated into pieces.
2. Merged cells cannot intersect.

The main cases that can lead to separation of merged cells are:

- Column or row insertion is not allowed if it can break a merged cell.
- Reordering of rows or columns (interactively or from code) is not allowed if the moved row/column belongs to a merged cell.
- If a column or row intersects a merged cell, the removal of this column or row is not allowed.
- The horizontal or vertical frozen area edges cannot cross merged cells, though merged cells can abut these lines by one of their sides.
- Merged cells cannot span across group rows or row text cells.

- If a grid contains merged cells spanning rows, sorting or grouping is not allowed by default as reordering rows leads to breaking merged cells in the general case.

Regarding intersection of merged cells, note that row text cells and group rows are also considered merged cells in this context and their intersections in various combinations are not allowed.

If the user is trying to do something that can lead to intersection of merged cells or their separation, nothing happens without a message for the user. For example, iGrid simply does not allow dragging of the column headers of the columns belonging to cells merged in the horizontal direction. If the developer is trying to do something in code that would separate a merged cell into pieces or lead to intersection of merged cells, iGrid raises an exception with the corresponding message.

Enabling sorting and grouping

Interactive sorting and grouping are disabled by default if iGrid contains cells merged in the vertical direction. However, if the logic of your application requires this functionality, you can enable these operations by unmerging the merged cells on the fly in the corresponding situations (for example, when the user clicks a column header to sort the grid by the corresponding column). This is done with the help of the **UnmergeCellsRequired** event.

To make sorting or grouping possible when the user clicks a column header or drags it to/from the group box area, iGrid raises the **UnmergeCellsRequired** event that allows you to unmerge cells in a handler of this event and tell iGrid about that so iGrid can proceed with sorting/grouping. To do this, you remove merging from the corresponding cells and do other required cleanup operations before sorting/grouping and notify iGrid about this work done by setting the **CellsUnmerged** property of the event arguments to True.

To unmerge cells in iGrid, you can use its **UnmergeAllCells** or **UnmergeCellsInCols** method. The simplest event handler that enables sorting in iGrid with merged cells looks like the following:

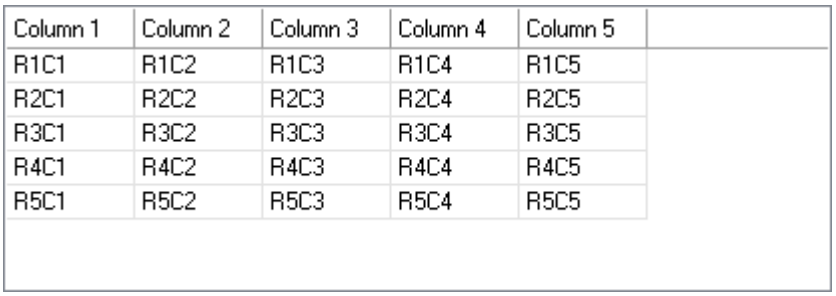
```
private void iGrid1_UnmergeCellsRequired(
  object sender, iGUnmergeCellsRequiredEventArgs e)
{
  iGrid1.UnmergeAllCells();
  e.CellsUnmerged = true;
}
```

Other properties of the event arguments represented with an object of the **iGUnmergeCellsRequiredEventArgs** class provides you with additional information about the event that requires cell unmerging to proceed. These are the operation type (the **Reason** property) and the column the operation is being performed for (the **ColIndex** property).

8. HEADER SECTION AND COLUMN HEADERS

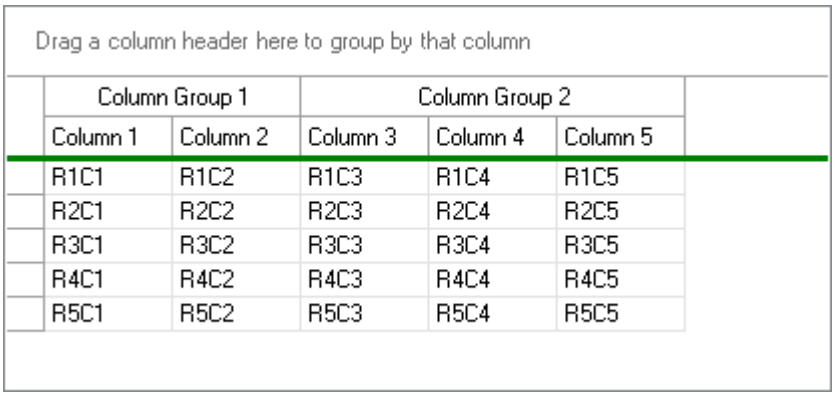
8.1. Overview of the Header Section Features

The iGrid header section is a rectangular area above the cell area, designed to display and manipulate column headers. In the simplest case it contains a row of column headers for visible columns:



Column 1	Column 2	Column 3	Column 4	Column 5
R1C1	R1C2	R1C3	R1C4	R1C5
R2C1	R2C2	R2C3	R2C4	R2C5
R3C1	R3C2	R3C3	R3C4	R3C5
R4C1	R4C2	R4C3	R4C4	R4C5
R5C1	R5C2	R5C3	R5C4	R5C5

In the general case the header section of iGrid may include other parts. The following screenshot demonstrates all available parts of the header section:



Drag a column header here to group by that column

	Column Group 1		Column Group 2		
	Column 1	Column 2	Column 3	Column 4	Column 5
	R1C1	R1C2	R1C3	R1C4	R1C5
	R2C1	R2C2	R2C3	R2C4	R2C5
	R3C1	R3C2	R3C3	R3C4	R3C5
	R4C1	R4C2	R4C3	R4C4	R4C5
	R5C1	R5C2	R5C3	R5C4	R5C5

The essential part of the header section is the area above normal cells displaying column headers for all visible columns. The extra empty non-clickable header cell next to the last column header is added by iGrid automatically if the viewport width is greater than the total width of all visible columns (see it after Column 5 on the screenshots). If row headers are displayed, the special clickable column header for the row header area is displayed at the left of the normal column headers (the empty column header before Column 1 on the second screenshot).

The general properties of the whole header section and the column headers it contains (appearance, ability to click column headers, colors for hot-track effects, etc.) are set with the **Header** object property of iGrid. One of the properties of the **Header** object, the Boolean **Visible** property, specifies the visibility of the header section and can be used to hide it:

```
iGrid1.Header.Visible = false;
```

If the user is allowed to group rows, the header section may display a special pane for manipulating column headers of groups. This pane is called "group box" and is located above column headers. You can see this pane with the hint "Drag a column header here..." on the second screenshot. The visibility of the group box is governed by the **Visible** property of the **GroupBox** object property of iGrid. The group box features are described in greater details in the Group Box and Interactive Grouping topic.

The grid lines framing column headers are constituent parts of the header section. The **HGridLinesStyle** and **VGridLinesStyle** properties of the **Header** object are used to configure them. The last horizontal grid line separating the header section from the cell area may have its own

style specified with the **SeparatingLine** property of the **Header** object. The second screenshot above demonstrates that this separating line was configured as a 3-pixel green line.

The header section contains one row of column headers after the grid control has been initialized. You can add more header rows if required — read the Managing Header Rows topic for more information. Header rows are numbered from bottom to top. For example, if you have a header section with 3 rows, the first row with the index of zero will be at the bottom of the header section and the last row with the index of 2 will be at the top of the header section.

An individual column header is represented with an instance of the **iGColHdr** class. A column header is similar to a normal cell with all its traditional properties — value, image, background and foreground colors, and so on. Because of this, column headers are also considered as column header cells. The value of a column header cell is displayed as the column title. In addition to normal cells, column header cells also implement some specific features, such as hot tracking effects or indication of sort status. All these features and the related properties are described in the series of Column Header Drawing Features topics.

Column header cells are organized in a 2-dimensional array. It is accessed with the **Cells** property of the **Header** object. This array is indexed by header row indexes and column indexes or their keys. In most cases, when the iGrid header has one row, you access the column headers in the **Cells** array using the zero row index. For example, the following statement sets the background color for the column header of the third column:

```
iGrid1.Header.Cells[0, 2].BackColor = Color.Yellow;
```

The same syntax can be used to change the title of the third column:

```
iGrid1.Header.Cells[0, 2].Value = "Column 3";
```

iGrid provides another, shorter way to set the column title — with the **Text** property of the column object (**iGCol**):

```
iGrid1.Cols[2].Text = "Column 3";
```

However, this approach can be used only to set the column titles of the first row in a multi-row header. And pay attention to the fact that the **Value** property of a column header cell has the universal **Object** type — while the **Text** property of the column object can accept only strings.

Column headers can be merged. This feature is often used in multi-row headers to merge adjacent columns into column groups or bands. You can see two column groups on the second screenshot above. The Merging Column Headers topic gives you more information on how this feature works in iGrid.

You can examine the source code below we used to create the grid on the second screenshot to see how all these concepts work together:

```
iGrid1.Cols.Add("Column 1");
iGrid1.Cols.Add("Column 2");
iGrid1.Cols.Add("Column 3");
iGrid1.Cols.Add("Column 4");
iGrid1.Cols.Add("Column 5");

iGrid1.GroupBox.Visible = true;
iGrid1.RowHeader.Visible = true;

iGrid1.Header.SeparatingLine.Width = 3;
iGrid1.Header.SeparatingLine.Color = Color.Green;

iGrid1.Header.Rows.Add();

iGrid1.Header.Cells[1, 0].Value = "Column Group 1";
iGrid1.Header.Cells[1, 0].SpanCols = 2;
iGrid1.Header.Cells[1, 0].TextAlign = iGContentAlignment.MiddleCenter;

iGrid1.Header.Cells[1, 2].Value = "Column Group 2";
iGrid1.Header.Cells[1, 2].SpanCols = 3;
iGrid1.Header.Cells[1, 2].TextAlign = iGContentAlignment.MiddleCenter;

iGrid1.Rows.Count = 5;
foreach (iGCell cell in iGrid1.Cells)
{
  cell.Value = String.Format("R{0}C{1}",
    cell.RowIndex + 1, cell.ColIndex + 1);
}
```

8.2. Column Header Drawing Features

8.2.1. Header Section Drawing Settings

The first thing that affects column header rendering on the screen is the settings of the whole header section. They are implemented as properties of the **Header** object property of iGrid.

Built-in header section styles

The **RenderStyle** property of iGrid defines the look of entire iGrid including its header area (see the Built-in Rendering Styles topic for more information). As a part of this concept, the iGrid header fully supports the three rendering styles — the system style, 3D, and flat. The header section, like other parts of iGrid, uses the system style and system colors by default. The following table demonstrates the built-in rendering styles available for the header section:

OS LOOK	3D LOOK	FLAT LOOK
		

Notice how different parts of the column header under the mouse pointer are highlighted depending on the used rendering style.

Header Grid Lines

The grid lines in the header section can be adjusted using the **HGridLinesStyle** and **VGridLinesStyle** properties of the **Header** object property. Some examples are below:

TRADITIONAL SOLID GRID LINES	DOTTED VERTICAL GRID LINES	WITHOUT GRID LINES
		

One more special property of the **Header** object, **SeparatingLine**, can be used to redefine the bottom grid line drawn below the whole header section. If the **Width** property of this line is greater than zero, this line is used at the bottom of the header section instead of the horizontal grid line defined with the **HGridLinesStyle** property of the **Header** object.

Note that the grid lines are not available when the header has the 3D appearance because of the specific look of column headers.

Header color properties

The default background color of the whole header section is determined by the theme used to render it — the OS visual styles, 3D, or flat look. If the header is drawn using the OS visual styles, the color comes from the OS visual styles. In the case of the 3D or flat look the default background color is **SystemColors**.**Control**. You can redefine the default background color of the header with the **BackColor** property of the **Header** object property. The same concerns the default background color of the hot and pressed header item — they can be overridden by custom values using the **HotTrackBackColor** and **PressedItemBackColor** properties of the **Header** object.

The **ForeColor** and **SortInfoColor** properties of the **Header** object property allow you to adjust the color of the column header titles and sort info.

Hot tracking effects

iGrid provides the hot tracking effect for column headers. The hot tracking effect can be adjusted separately for every visual theme of the header (the OS styles, 3D, and flat look). The following properties of the **Header** object are used to adjust the hot tracking effect parameters in iGrid column headers:

PROPERTY	DESCRIPTION
HotTracking	Gets or sets a Boolean value indicating whether hot tracking effects are available in the header. Set it to False to prohibit any hot tracking effects; in this case the values of all properties listed below are ignored.
HotTrackFlagsVisualStyle	Gets or sets a value indicating which parts of the header cell should indicate the hot state when the header has the OS visual style. A combination of the flags from the **iGHdrHotTrackFlags** enumeration.
HotTrackFlagsClassicStyle	Gets or sets a value indicating which parts of the header cell should indicate the hot state when the header has the 3D appearance. A combination of the flags from the **iGHdrHotTrackFlags** enumeration.
HotTrackFlagsFlatStyle	Gets or sets a value indicating which parts of the header cell should indicate hot state when the header has the flat appearance. A combination of the flags from the **iGHdrHotTrackFlags** enumeration.

PROPERTY	DESCRIPTION
HotTrackBackColor	Gets or sets the background color of the hot header cell. Has an effect only if the **Background** flag is specified in the **HotTrackFlagsVisualStyle**, **HotTrackFlagsClassicStyle** or **HotTrackFlagsFlatStyle** property corresponding to the current look.
HotTrackForeColor	Gets or sets the color of the text in the hot header cell. Has an effect only if the **Text** flag is specified in the **HotTrackFlagsVisualStyle**, **HotTrackFlagsClassicStyle** or **HotTrackFlagsFlatStyle** property corresponding to the current look.
HotTrackIconDegree	Gets or sets a value indicating how the icon in the hot header cell should be highlighted — lightened or darkened depending on the value of this property (possible values are integers from -100 to 100). Has an effect only if the **Icon** flag is specified in the **HotTrackFlagsVisualStyle**, **HotTrackFlagsClassicStyle** or **HotTrackFlagsFlatStyle** property corresponding to the current look.

Note that the hot tracking effect can be disabled for the whole iGrid with its **HotTracking** property. In this case the hot tracking effect in the header is automatically turned off regardless of the value of the **Header**.**HotTracking** property.

8.2.2. Formatting Individual Column Headers

An individual column header is represented with an object of the **iGColHdr** class. Generally you retrieve these objects when you access the **Cells** collection of the **Header** object of iGrid:

```
iGrid1.Header.Cells[0, 2].BackColor = Color.Pink;
```

A column header can be formatted either through its style (the **Style** property) or through its properties (**BackColor**, **ForeColor**, ...). In the latter case you actually change the style of the column header. If the style of the column header has not been created, it is created automatically when you set one of these properties. This works the same way like for iGrid cells.

The following table lists the properties of the iGrid column header that define its look and behavior:

PROPERTY	DESCRIPTION
BackColor	The background color of the column header.
ContentIndent	Defines the left, top, right and bottom indent of the column header contents.
CustomDrawFlags	Defines which column header parts should be drawn by the developer.
DropDownControl	Determines whether the column header contains an attached drop-down control and the combo button to open it.
Flags	Gets or sets flags that determine which parts of the column header's contents (image, text) are displayed.
Font	The font used to display the column header text.

PROPERTY	DESCRIPTION
ForeColor	The foreground color of the column header (its text).
FormatProvider	Contains the reference to an object that implements the **IFormatProvider** interface. Can be used to implement a complex formatting for the column header when the **FormatString** property is not enough.
FormatString	Defines the format string applied to the column header value before it is displayed on the screen (like in the **String**.**Format** method).
ImageAlign	The horizontal and vertical alignment of the image in the column header.
ImageList	Gets or sets the image list that contains the images to display in the column header.
SortInfoVisible	Get and sets the value indicating whether the sort info is visible. Sort info is the sort arrow and sort index (for multicolumn sorting). Use this property for narrow columns.
TextAlign	The horizontal and vertical alignment of the column header text.
TextFormatFlags	Defines the column header text format flags (looks like the **StringFormatFlags** enumeration in the .NET Framework).
TextPosToImage	The relative position of the column header icon and text.
TextTrimming	Defines the column header text trimming options (like the **StringTrimming** enumeration in the .NET Framework).

Column headers support style inheritance like cells too. The **Style** property of a column header is not initialized by default. In this case iGrid.NET uses another column header style object stored in each iGrid column. It is the **ColHdrStyle** object property of a column object (**iGCol**). This column header style is created every time when you create a new column, and its properties are used to format the column header(s) for this column.

Any property of a column header style can be set to the special 'NotSet' value. This value depends on the type of the property. For object types it is null (Nothing in VB), for color properties — the special **Color**.**Empty** value, for enum properties — the NotSet value from the enum and so on. If a property of the column header style has the NotSet value, iGrid looks for the same property in the header style object for the whole column (**ColHdrStyle**).

If a property of a **ColHdrStyle** object has the NotSet value too, iGrid determines the effective property value according to the property. For properties that have equivalents in the **Header** object (like **Font** or **ForeColor**), their values are retrieved from the corresponding properties of the header. Other properties, such as **TextAlign**, are considered to have the default values. For example the default value for **TextAlign** is **MiddleLeft**. This inheritance of column header properties allows you to change the formatting of the whole header easily when you change just the properties of the **Header** object.

8.2.3. Custom-Drawn Column Headers

In many cases iGrid column headers work like normal cells, and custom drawing is not an exclusion. Like any iGrid cell, every column header can have a custom-drawn background or/and foreground. To use custom drawing in a column header, do the following:

1. Set the **CustomDrawFlags** property of the column header's style to a set of the **iGCustomDrawFlags** values.
2. Add an event handler to the **CustomDrawColHdrBackground** and/or **CustomDrawColHdrForeground** events.
3. Add an event handler to the **CustomDrawColHdrGetWidth** and **CustomDrawColHdrGetHeight** events if needed.

Custom drawing in column headers is very similar to custom drawing in normal cells. The only difference is that a column header can display sort info. Sort info is the arrow and sort index displayed when iGrid is sorted by a column. The location of sort info depends on the **ImageAlign** and **TextAlign** properties of a column header. If a column header's image and text are left aligned, then its sort info is drawn just after the text and image; otherwise the sort info will be aligned to the right edge of the column header.

If the **TextAlign** and **ImageAlign** properties are set to their default values (left-top alignment), and the sort info should be drawn just after the column header's text and image, you draw your custom foreground in the **CustomDrawColHdrForeground** event and iGrid determines the horizontal position of the sort info based on the width of your custom foreground. To tell iGrid the width of your custom column header contents, write the corresponding event handler for the **CustomDrawColHdrGetWidth** event.

In the rest of cases (when the sort info is drawn at the right edge), the **CustomDrawColHdrForeground** event provides you with the bounds of the column header excluding the sort info area, grid lines and the current system style padding from it.

8.3. Managing Header Rows

The iGrid header has one column header row after initialization. You cannot remove it, but you can add more rows if required.

The simplest way to add additional rows to the only default header row is increasing the **Count** property of the **Rows** property of the **Header** object. For example, the following statement will add a new row with empty column headers above the default header row:

```
iGrid1.Header.Rows.Count = 2;
```

Another way to create new header rows is to call the **Add** or **AddRange** method of the **Header**.**Rows** collection object. Remember that the rows in the header are indexed from bottom to top in contrast to the normal rows, so that new rows are added to the top of the header.

The following line of code demonstrates how to create a grid with two rows in its header (we imply that the header already has one row created by default):

```
iGrid1.Header.Rows.Add();
```

When you add a new row to the header, its properties are initialized to the corresponding default values. You can change the default values for some header row properties when you add a new row using the overloaded versions of the **Add** and **AddRange** methods of the **Header**.**Rows** collection. Use the overloaded method versions that accept the **iGHdrRowPattern** class as their parameter. For example, the following code snippet adds two more rows with the height of 40 pixels to the header:

```
iGHdrRowPattern hdrRowPattern = new iGHdrRowPattern();
hdrRowPattern.Height = 40;
iGrid1.Header.Rows.AddRange(2, hdrRowPattern);
```

Individual iGrid header rows can be accessed with the same **Rows** collection. This collection is indexed by integer row indexes starting with zero. Access to the properties of a header row is provided by the **iGHdrRow** class. Instances of this class are returned by the **Rows** collection when you retrieve its items. Below is an example demonstrating how to change the visibility of the third header row:

```
iGrid1.Header.Rows[2].Visible = false;
```

To remove a header row or a range of rows, call the **RemoveAt** or **RemoveRange** methods of the **Header**.**Rows** collection or decrease the **Header**.**Rows**.**Count** property. Remember that the very first header row (right above cells) cannot be removed.

The **Header**.**Rows** collection provides the useful multi-purpose **Reset** method. It removes all existing header rows except the first one, splits all merged column headers, and clears the contents of all column headers at one go. Consider the usage of this method in scenarios in which you need to remove all header rows.

8.4. Setting Header Row Heights

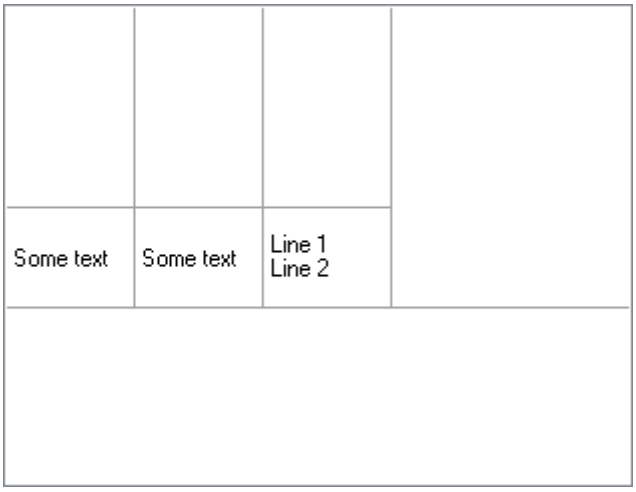
If you are using iGrid with the default header settings, the heights of header rows are automatically adjusted when one of the following events occurs:

- a column is added;
- a column is removed;
- the contents or formatting of a column header is changed;
- the rendering style of the header is changed.

In most real-world situations this helps you to avoid writing code in which you calculate and set optimal header row heights to display column header contents. However, this feature may prevent you from setting the desired header row height in some cases. Let's consider the following code sample:

```
iGrid1.Cols.Count = 3;
iGrid1.Header.Rows.Count = 2;

iGrid1.Cols[0].Text = "Some text";
iGrid1.Cols[1].Text = "Some text";
iGrid1.Cols[2].Text = "Line 1\r\nLine 2";

iGrid1.Header.Rows[0].Height = 50;
iGrid1.Header.Rows[1].Height = 100;
```

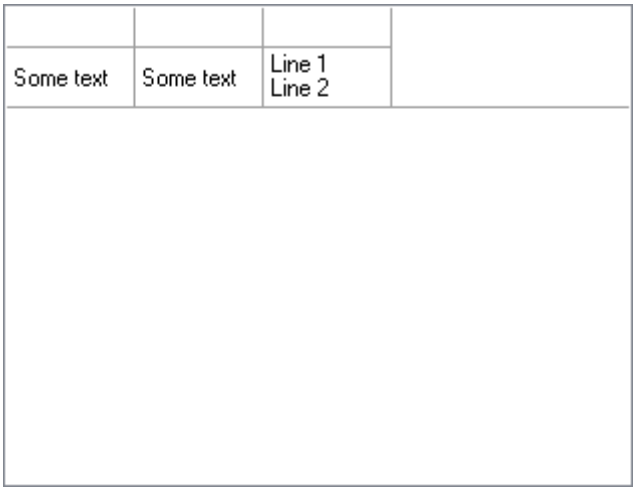
It creates a grid with two header rows with the specified heights — 50 and 100 pixels:



However, if we set header row heights before setting column titles like in the following code

```
iGrid1.Cols.Count = 3;
iGrid1.Header.Rows.Count = 2;

iGrid1.Header.Rows[0].Height = 50;
iGrid1.Header.Rows[1].Height = 100;

iGrid1.Cols[0].Text = "Some text";
iGrid1.Cols[1].Text = "Some text";
iGrid1.Cols[2].Text = "Line 1\r\nLine 2";
```

, we will see not the result we expect:



The problem is that iGrid will automatically adjust the header row heights when you set column titles. This behavior is governed by the **AutoHeightEvents** property of the **Header** object. It contains a set of flags from the **iGAutoHeightEvents** enumeration that define the cases in which the header row heights must be adjusted automatically. The default value is the combination of the flags **OnAddCol**, **OnRemoveRow**, **OnContentsChange**, and **OnThemeChange** — which corresponds to the events listed above.

To fix the problem with setting specific header row heights before we set column titles, we can completely turn off automatic adjustment of header row heights by setting the **Header**.**AutoHeightEvents** property to **None**:

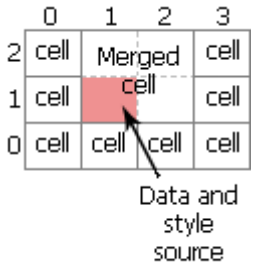
```
iGrid1.Cols.Count = 3;
iGrid1.Header.Rows.Count = 2;

iGrid1.Header.AutoHeightEvents = iGAutoHeightEvents.None;

iGrid1.Header.Rows[0].Height = 50;
iGrid1.Header.Rows[1].Height = 100;

iGrid1.Cols[0].Text = "Some text";
iGrid1.Cols[1].Text = "Some text";
iGrid1.Cols[2].Text = "Line 1\r\nLine 2";
```

You can automatically fit the height of an individual header row by calling its **AutoHeight** method (see the **iGHdrRow** class). You can also adjust the height of all header rows at once with the **AutoHeight** method of the **Header** object.

8.5. Merging Column Headers

The SpanRows and SpanCols properties of a column header

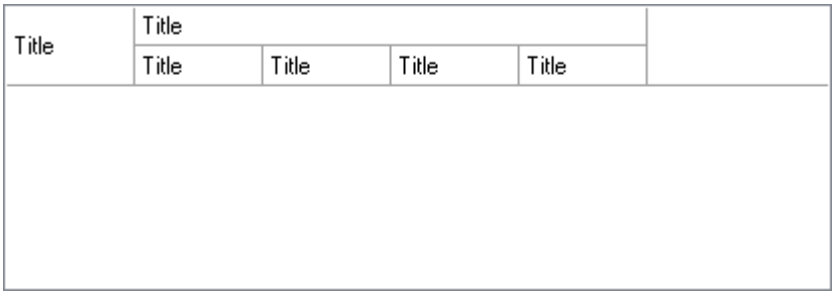
Adjacent header cells can be merged into one. The behavior, appearance and contents of a merged header cell is defined by the properties of the bottom left header cell belonging to this group:



To merge header cells, use the **SpanRows** and **SpanCols** properties of the **iGColHdr** class. The following example shows how to create a grid with 2-row header and merge header cells:

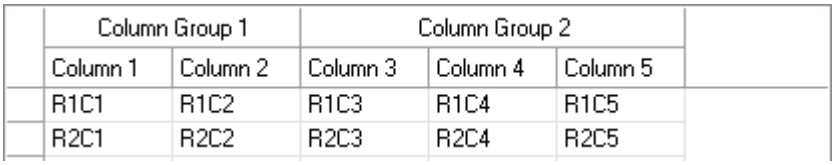
```
private void Form1_Load(object sender, System.EventArgs e)
{
  // Set the default title.
  iGrid1.DefaultCol.Text = "Title";

  // Add 5 columns.
  iGrid1.Cols.AddRange(5);

  // Add the second header row.
  iGrid1.Header.Rows.Add();

  // Merge column headers.
  iGrid1.Header.Cells[0, 0].SpanRows = 2;
  iGrid1.Header.Cells[1, 1].SpanCols = 4;
}
```

The result is on the picture below:

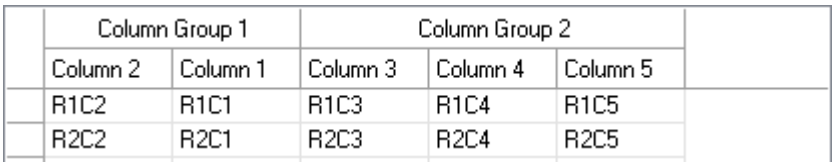


Restrictions in grids with merged column headers

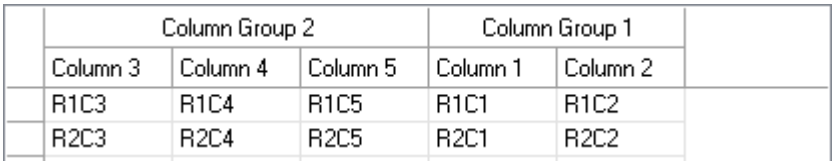
Columns with merged headers can be moved only within the bounds of the merged area. All the columns belonging to one merged area can be moved to another place as a whole group. Look at the following sample of the iGrid header with merged column headers in the second row:



In this grid you will be able to move the first column only to the second position (after "Column 2") and vice versa:



You will also be able to move the first and second columns as one group to the last position if you drag their merged column header ("Column Group 1") in the top header row:



If you already have merged column headers in iGrid, you cannot set the number of frozen columns to a value so that a merged column header would be separated by the frozen columns area edge. You also cannot move a group of columns with merged headers to the position that contains the frozen columns area edge to separate merged headers.

8.6. Header Section Mouse Events

iGrid provides you with a rich set of mouse events related to its header section. All these mouse events are triggered for column headers, and almost all names of these events start with 'ColHdr'.

The main traditional mouse events generated for column headers are: **ColHdrMouseDown**, **ColHdrMouseMove**, **ColHdrMouseUp**, **ColHdrMouseEnter** and **ColHdrMouseLeave**. The arguments of these events provide you with the information about the column header (its column index, bounds), the location of the mouse pointer, the pressed mouse button and the like.

However, these are low-level mouse events and they are used rarely enough. To make processing of user actions like column header click or column header dragging easier, iGrid raises other events related to column headers. Among them the **ColHdrClick**, **ColHdrStartDrag**, **ColHdrDragging** and **ColHdrEndDrag** events with self-descriptive names. You can also process double clicks on column headers or their dividers using the **ColHdrDoubleClick** or **ColDividerDoubleClick** events respectively.

The two typical properties of the event arguments objects of these events are **ColIndex** and **Kind**. The first of them, **ColIndex**, allows you to know the index of the column for which the event was triggered. The second property, **Kind** of the **iGColHdrKind** enumeration type, is used to know the type of the column header. If you want to access the grid column by the **ColIndex** value, first you must check the value of **Kind** to know whether the event was generated for a real column. The fact is that most of the column header events listed above are triggered not only for headers of columns with normal grid cells, but also for the clickable column header for the row header area (if it is visible) and the extra header cell after the last column header. The **ColIndex** property contains a valid column index only if the event was raised for a real column, which corresponds to the **Normal** value of the **Kind** property of the event arguments object.

Thus, a proper event handler for processing clicks in column headers for columns with cells should look like this:

```
void iGrid1_ColHdrClick(object sender, iGColHdrClickEventArgs e)
{
   if (e.Kind == iGColHdrKind.Normal)
   {
      // column header click for a normal column
   }
}
```

And if you want to process clicks in the column header for the row header area, you should use an event handler like the following one:

```
void iGrid1_ColHdrClick(object sender, iGColHdrClickEventArgs e)
{
   if (e.Kind == iGColHdrKind.RowHdr)
   {
      // clicking the row header area's column header
   }
}
```

Many column header events also allow you to disable the corresponding user action. It is done with the help of the **DoDefault** property of the event arguments. For example, you can disable sorting of a column when its column header is clicked in an event handler of the **ColHdrClick** event by setting the **DoDefault** property of its event arguments to False. Similarly, you can disable the automatic adjustment of the column width when the user double-clicks its column divider with the **ColDividerDoubleClick**, and so on.

The column header mouse events work the same way for column headers when they are placed inside the group box. The event arguments objects of some events even provide an additional property allowing you to determine whether the column header is currently inside the group box. For example, this is the Boolean **ToGroupBox** property of the event arguments objects of the **ColHdrDragging** and **ColHdrEndDrag** events. If a column header event does not provide you with such a flag, you can always determine whether the column header is in the group box by finding the corresponding column in iGrid's **GroupObject** (see its **Contains** method).

One note regarding column header tooltips. They are displayed as a reaction to events related to the movement of the mouse cursor. You can customize column header tooltips or set custom tooltip texts with the **RequestColHdrToolTipText** event. This event is a part of the built-in tooltip system in iGrid and is described in greater details in the Built-in Tooltips for Cells of All Kinds topic.

8.7. Protecting Header from User Changes

The developer may need to disable interactive changes in the header area. iGrid provides you with many members that affect the header functionality, but they are not always properties and methods

of the **Header** object property. Because of this, sometimes it is hard to find the required iGrid member to implement a task related to the header. The aim of this topic is to help you to find the related members when you need to prohibit some user actions in the header.

One of the typical tasks related to the header is disabling clicks on column headers to prevent iGrid from sorting. This can be implemented with the **Header**.**AllowPress** property:

```
iGrid1.Header.AllowPress = false;
```

The same functionality can be implemented with the **ColHdrClick** event. The **ColHdrClick** event allows you to disable clicking on both all and individual column headers. Below is an example of the event handler to disable clicks only on the column header of the first column:

```
private void iGrid1_ColHdrClick(
  object sender, iGColHdrClickEventArgs e)
{
  if (e.ColIndex == 0)
    e.DoDefault = false;
}
```

Disabling clicks on column headers does not prevent them from being reordered by the user. To disable moving of a column, set its **AllowMoving** property to False. This setting can be applied to all columns with the **DefaultCol** object of iGrid before you create any columns:

```
iGrid1.DefaultCol.AllowMoving = false;
```

As it is generally true for iGrid, you can achieve the same effect with the corresponding event — the **ColHdrStartDrag** event in this case:

```
private void iGrid1_ColHdrStartDrag(
  object sender, iGColHdrStartDragEventArgs e)
{
  e.DoDefault = false;
}
```

Like in the case of the **ColHdrClick** event mentioned above, you can disable column movement for individual columns using the arguments of the **ColHdrStartDrag** event.

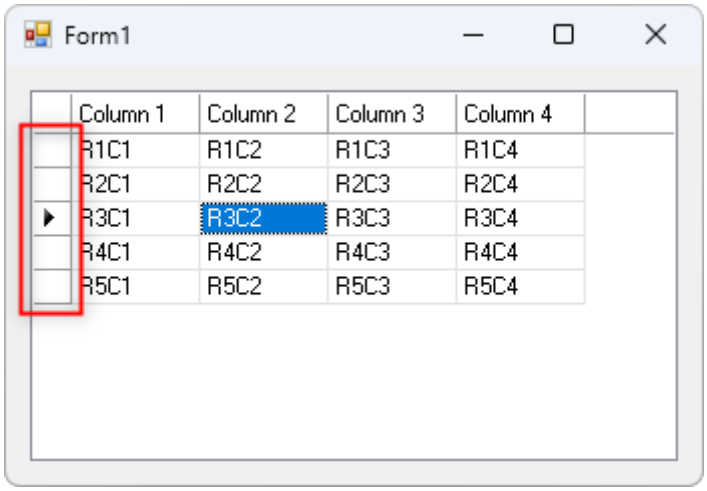
Interactive resizing of columns is disabled with the **AllowSizing** property for individual columns or with the same property of the **DefaultCol** object of iGrid before you create any columns.

If the group box is visible, the user has the ability to drag column headers into that area to group iGrid by the corresponding columns. You can disable this action for individual columns using their **AllowGrouping** property.

9. ROW HEADERS

9.1. Row Header Area and Its Properties

The row header area is an optional area before grid rows designated to display a special row header element for every row. Row headers are used to indicate the current row; they also help to select rows and resize them. If the row header area is visible, it is always present in the viewport and is not scrolled in the horizontal direction together with row contents. The row header area is marked with a red rectangle on the picture below:



The appearance and behavior of the whole row header area is configured with the **RowHeader** object property of iGrid. The main property of this object is **Visible**. It allows you to show the row header area that is hidden by default:

```
iGrid1.RowHeader.Visible = true;
```

The other main properties of the **RowHeader** object defining the look and behavior of this area are:

PROPERTY	DESCRIPTION
BackColor	Determines the background color of the row header area.
GlyphColor	Determines the color used to draw row header glyphs that are not related to errors.
GlyphErrorCircleColor	Determines the color of the filled circle used in row header glyphs related to errors.
GlyphErrorMarkColor	Determines the color of the mark drawn inside row header glyphs related to errors.
HGridLinesStyle	Determines the style of the horizontal grid lines.
HotTrackBackColor	Determines the background color of the hot row header. The default value is **Color.Empty**, meaning that the hot tracking effect from the current theme is used.
HotTracking	Determines whether row headers should indicate the hot state.

PROPERTY	DESCRIPTION
HotTrackBackColor	Determines the background color of the pressed item. The default value is **Color**.**Empty**, meaning that the hot tracking effect from the current theme is used.
VGridLinesStyle	Determines the style of the vertical grid line. This line separates row headers from normal grid cells.
Visible	Determines whether the row header area is visible.
Width	Determines the width of the row header area.
WidthAutoSet	Specifies whether the width of the row header area is automatically set to its optimal value after the control initialization.

The **RenderStyle** property of iGrid defines the look of iGrid including the style of its row header area. You can find out more about the built-in rendering styles in iGrid from the Built-in Rendering Styles topic.

9.2. Row Header Area Drawing Settings

One of the key ideas of iGrid is to provide the look and fill consistent with the OS. As a part of this concept, the iGrid row header area supports the three main rendering styles — the system style, 3D style, and flat style. The row header area, like other parts of iGrid, automatically uses the system style and system colors by default (see the Built-in Rendering Styles topic for more information).

The drawing concept of the row header area mimics the drawing concept of the column header area because in this context row headers are column headers but for rows. All details of the header drawing concept can be found in the Header Section Drawing Settings topic in this manual. This topic briefly describes the main properties related to the row header area drawing.

The following table demonstrates how the row header area looks when it uses one of the built-in rendering styles:

SYSTEM STYLE	3D STYLE	FLAT STYLE
		

Notice how the row header under the mouse pointer is highlighted depending on the used rendering style.

The default background color of the whole row header area is determined by the theme used to render it. If the row header area is drawn using the OS visual styles, the color comes from the OS visual styles. In the case of the 3D or flat look the default background color is **SystemColors**.**Control**. You can redefine the default background color of the row header area with the **BackColor** property of the **RowHeader** object property. The same concerns the default background color of the hot and pressed row header item — they can be overridden by custom values using the **HotTrackBackColor** and **PressedItemBackColor** properties of the **RowHeader** object. To highlight an individual row header with a color that differs from the row header area color, use the **RowHdrDynamicBackColor** event of iGrid.

If row headers are not rendered with the 3D style, their rendering is similar to the rendering of normal cells and you have the ability to adjust the grid lines framing every row header. This is done with the help of the **HGridLinesStyle** and **VGridLinesStyle** properties of the **RowHeader** object property of iGrid.

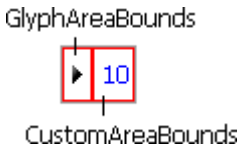
9.3. Custom-Drawn Row Headers

As with cells and column headers, iGrid allows you to draw custom content in row headers. You can draw custom row header background and foreground by handling the **CustomDrawRowHdrBackground** and **CustomDrawRowHdrForeground** events of iGrid. Unlike cells and column headers, row headers do not need to be explicitly marked as custom-drawn by setting a property, because these events are raised for every row header automatically. When necessary, you can use the **DoDefault** parameter of these events to suppress drawing of the standard row header elements.

The **CustomDrawRowHdrBackground** event has only one coordinate-related parameter, **Bounds**. It provides the bounds of the row header background area, that is, the row header bounds with the grid lines excluded.

The **CustomDrawRowHdrForeground** event has two parameters that define the bounds of the row header areas being drawn: **GlyphAreaBounds** and **CustomAreaBounds**. GlyphAreaBounds specifies the area where the standard row header glyph is drawn, whereas CustomAreaBounds specifies the remaining area available for other contents.

The **CustomDrawRowHdrForeground** event has two parameters that determine the coordinates of the row header being drawn: **GlyphAreaBounds** and **CustomAreaBounds**. The former provides the bounds of the area where the standard row header glyph is drawn. The latter provides the bounds of the remaining area available for other contents:



The following example shows how to draw row numbers on the right of the standard row header glyph:

```
StringFormat fRowHeaderStringFormat = new StringFormat();

private void Form1_Load(object sender, EventArgs e)
{
  fRowHeaderStringFormat.Alignment = StringAlignment.Far;
  fRowHeaderStringFormat.LineAlignment = StringAlignment.Center;

  iGrid1.RowHeader.Visible = true;
  iGrid1.RowHeader.Width = 30;

  iGrid1.Cols.Count = 3;
  iGrid1.Rows.Count = 10;

  iGrid1.CustomDrawRowHdrForeground += iGrid1_CustomDrawRowHdrForeground;
}

private void iGrid1_CustomDrawRowHdrForeground(
  object sender, iGCustomDrawRowHdrForegroundEventArgs e)
{
  Rectangle myTextBounds = e.CustomAreaBounds;
  if (myTextBounds.Width > 0 && myTextBounds.Height > 0)
  {
    e.Graphics.DrawString(e.RowIndex.ToString(),
      iGrid1.Font, Brushes.Blue, myTextBounds, fRowHeaderStringFormat);
  }
}
```

The result is on the screenshot below:

If you draw custom content in row headers, you should also handle the **CustomDrawRowHdrGetHeight** and **CustomDrawRowHdrGetWidth** events. These events are raised when iGrid automatically adjusts row height or row header width to fit the contents.

In a handler of the **CustomDrawRowHdrGetHeight** event, specify the heights of the glyph and custom areas shown in the scheme above by using the **GlyphAreaHeight** and **CustomAreaHeight** parameters, respectively. The final row height is calculated as the greater of these two values plus the horizontal grid line width and the system border indents. In other words, you only need to specify the height of the contents you want to display, and iGrid automatically takes all other factors into account.

Similarly, the **CustomDrawRowHdrGetWidth** event allows you to specify the widths of the glyph and custom areas by using the **GlyphAreaWidth** and **CustomAreaWidth** parameters. When calculating the final row header width, iGrid adds these two values together and then adds the vertical grid line width and the system border indents.

If you need to draw a standard row header glyph, use the **DrawRowHdrGlyph** method of iGrid. See also the Row Header Glyphs topic.

9.4. Row Header Glyphs

Row header glyphs are small visual indicators displayed in row headers to reflect the state of a row or provide additional information about it. For example, a glyph can show that the row is the current row, that a cell in the row is being edited, or that the row contains an error.

In iGrid, the **iGRowHdrGlyph** enumeration defines all available row header glyphs:

- **CurRow** — indicates the current row.
- **Editing** — indicates that a cell in the row is currently being edited.
- **NewRow** — indicates the special new-row placeholder used to add a new row.
- **UncommittedChanges** — indicates that the row contains changes that have not yet been committed.
- **Error** — indicates that the row contains an error.
- **CurRowError** — indicates that the row is the current row and also contains an error.
- **None** — indicates that no glyph is displayed.

You can determine which glyph is currently displayed in a row header using the **HdrGlyph** property of the corresponding row object (**iGRow**).

iGrid automatically draws either the **CurRow** or **Editing** glyph in the row header of the current row, depending on whether the grid is in browse mode or edit mode. The other built-in glyphs can be drawn by calling the **DrawRowHdrGlyph** method of iGrid in its **CustomDrawRowHdrForeground** event handler. The following example shows how to draw the **NewRow** glyph in the row header of the row used to add a new row:

```
private void fGrid_CustomDrawRowHdrForeground(
  object sender, iGCustomDrawRowHdrForegroundEventArgs e)
{
  if (IsAddNewCustomerRow(e.RowIndex))
  {
    fGrid.DrawRowHdrGlyph(
      e.Graphics,
      iGRowHdrGlyph.NewRow,
      e.GlyphAreaBounds.X, e.GlyphAreaBounds.Y,
      e.GlyphAreaBounds.Width, e.GlyphAreaBounds.Height);
    e.DoDefault = false;
  }
}
```

iGrid allows you to customize the default colors used to draw row header glyphs. This is done by using the following properties:

PROPERTY	DESCRIPTION
GlyphColor	Determines the color used to draw row header glyphs that are not related to errors.
GlyphErrorCircleColor	Determines the color of the filled circle used in row header glyphs related to errors.
GlyphErrorMarkColor	Determines the color of the mark drawn inside row header glyphs related to errors.

These properties default to **Color**.**Empty**. When left at this value, iGrid automatically uses the colors defined by the current theme.

9.5. Automatic Adjustment of Row Header Area Width

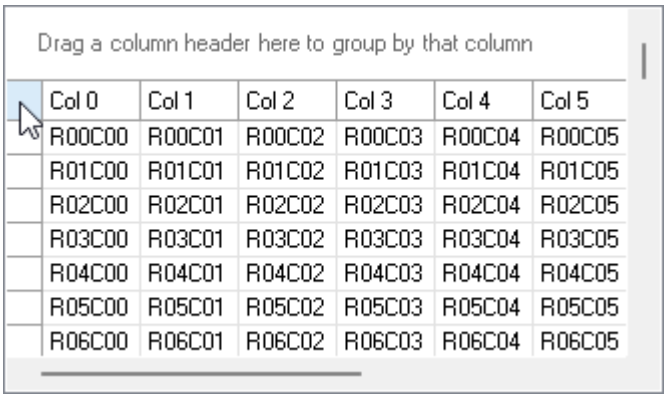
By default, iGrid automatically sets the width of the row header area to the optimal value upon completion of initialization in the code generated by the Windows Forms designer. The optimal width considers the screen DPI and makes sure the row header glyphs are fully visible without clipping. This behavior is controlled by the Boolean **WidthAutoSet** property of the **RowHeader** object property of iGrid. The default value of **WidthAutoSet** is True, which enables this automatic width adjustment; set it to False to disable it.

This automatic width adjustment is performed in the **EndInit** method, which is called automatically by the Windows Forms designer after control initialization. If the **WidthAutoSet** property is set to True, iGrid sets the **Width** property of **RowHeader** to the value returned by the **GetPreferredWidth** method of **RowHeader** within the **EndInit** method. The **GetPreferredWidth** method returns the optimal width of the row header area, and you can also call it from your code when you need to know the optimal row header width.

iGrid also lets you adjust the width of the row header area manually when needed. This is typically useful if you use custom drawing in row headers. Call the **AutoWidth** method of the **RowHeader** object property of iGrid to automatically adjust the width of the row header area. If the row header contains custom content, you should also handle the **CustomDrawRowHdrGetWidth** event to tell iGrid the width of row header with custom contents.

9.6. Clickable Header for Row Header Area

If the row header and column header areas are visible, iGrid displays an empty column header above the row header area. You can see this special column header on the following screenshot as highlighted under the mouse pointer:



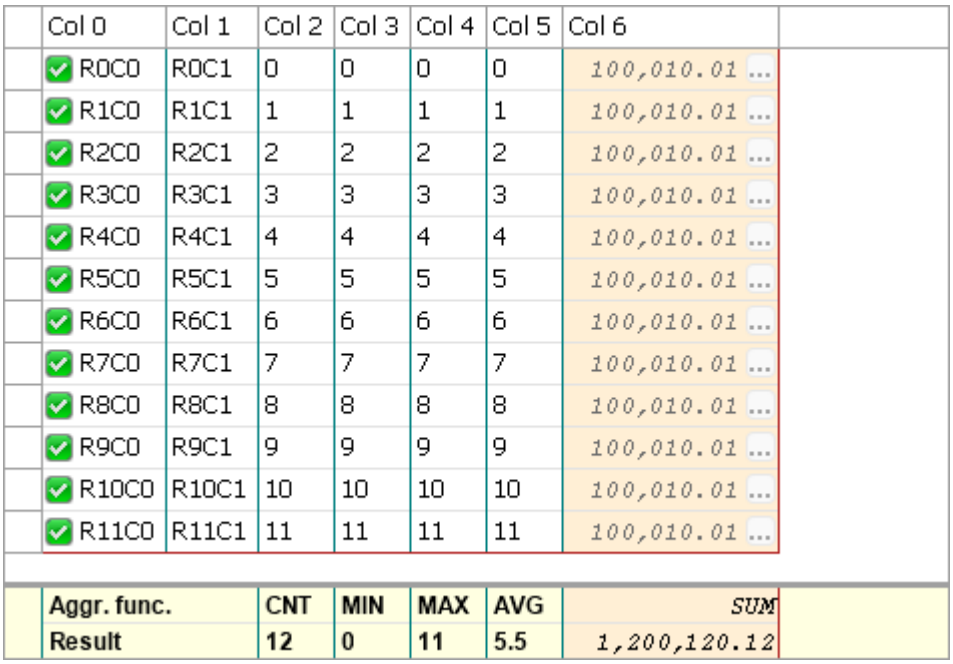
This element is clickable, and you can use the **ColHdrClick** event to process the clicks on it to do something useful. For instance, many spreadsheet applications like Microsoft Excel use it to select all cells when the user clicks it. In iGrid, this can be implemented using an even handler like this:

```
private void iGrid1_ColHdrClick(object sender, iGColHdrClickEventArgs e)
{
  if (e.Kind == iGColHdrKind.RowHdr)
    iGrid1.PerformAction(iGActions.SelectAllCells);
}
```

10. FOOTER SECTION

10.1. General Features of Footer

The footer section is a special non-scrollable area that can be displayed at the bottom of the grid control. It has the same set of columns defined in the grid and can have one or several rows. As a rule, the footer is used to display totals for data in the normal cells:



	Col 0	Col 1	Col 2	Col 3	Col 4	Col 5	Col 6
	R0C0	R0C1	0	0	0	0	100,010.01 ...
	R1C0	R1C1	1	1	1	1	100,010.01 ...
	R2C0	R2C1	2	2	2	2	100,010.01 ...
	R3C0	R3C1	3	3	3	3	100,010.01 ...
	R4C0	R4C1	4	4	4	4	100,010.01 ...
	R5C0	R5C1	5	5	5	5	100,010.01 ...
	R6C0	R6C1	6	6	6	6	100,010.01 ...
	R7C0	R7C1	7	7	7	7	100,010.01 ...
	R8C0	R8C1	8	8	8	8	100,010.01 ...
	R9C0	R9C1	9	9	9	9	100,010.01 ...
	R10C0	R10C1	10	10	10	10	100,010.01 ...
	R11C0	R11C1	11	11	11	11	100,010.01 ...
	Aggr. func.		CNT	MIN	MAX	AVG	SUM
	Result		12	0	11	5.5	1,200,120.12

The footer section is controlled with the **Footer** object property of iGrid. It is an object of the **iGFooter** class. Its properties and methods allow you to configure the footer section.

By default the footer section is invisible in every new instance of iGrid. To make the footer section visible, set its **Visible** property to True:

```
iGrid1.Footer.Visible = true;
```

The footer section has one row by default. You can create additional rows in the footer with the **Rows** property of the **Footer** object. Read the Footer Rows topic for more information.

The main footer formatting properties are **BackColor**, **ForeColor**, **Font**. By default the footer uses the colors of the system Windows tooltip. The **Font** property of the footer is not set by default, which means the font of the whole grid is used (the **Font** property of iGrid).

The whole footer section is separated from the area of normal cells with the special horizontal line at the top of the footer section. The properties of this separating line (thickness, color and style) can be set with the **SeparatingLine** property of the **Footer** object.

The footer does not have its own grid line settings. The styles of the vertical and horizontal grid lines in the footer are inherited from the normal cell area (see the Cell Grid Lines topic for more details).

10.2. Footer Rows

Creating and removing footer rows

The collection of footer rows can be accessed via the **Rows** property of the **Footer** object. It returns an instance of the **iGFooterRowCollection** class. Its properties and methods are used to manipulate footer rows. For example, you can add two more footer rows to the single default footer row with the following statement:

```
iGrid1.Footer.Rows.Count = 3;
```

Footer rows are indexed from top to bottom. For instance, you can access the 3rd row as **iGrid.Footer.Rows[2]**. An expression like this returns an object of the **iGFooterRow** class that represents an individual footer row. The two main properties of **iGFooterRow** are **Visible** and **Height**. They are used to show/hide a particular footer row and to change its height respectively.

You can remove existing footer rows by decreasing the **Count** property of the **Footer**.**Rows** collection or by calling its **RemoveAt** or **RemoveRange** methods. Pay attention to the fact that the footer section always has one non-removable row, like the header section. Therefore, you cannot remove the footer row with index 0.

The **Footer**.**Rows** collection provides the **Reset** method that deserves a mention. This method is used to reset the iGrid footer to its default state, i.e. one footer row with empty footer cells. In other words, it can be used to remove all footer rows except the first one and to clear the contents of footer cells in the remaining row.

Adjusting the height of footer rows

By default the height of every footer row is adjusted automatically when you change the contents of its cells or related formatting properties like font. This functionality is controlled by the flags specified in the **AutoHeightEvents** property of the **Footer** object. To disable the automatic height adjustment for footer rows, remove the corresponding flags from the **AutoHeightEvents** property or simply set it to **None** before any operation with the footer:

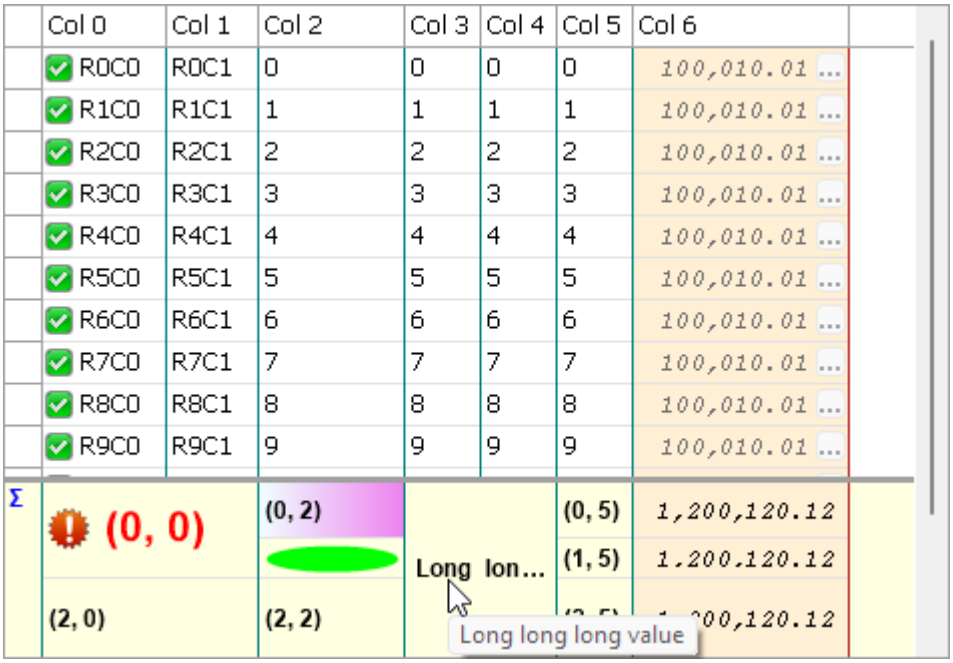
```
iGrid1.Footer.AutoHeightEvents = iGAutoHeightEvents.None;
```

Even if this setting is in effect, you can still automatically adjust the height of a footer row with its **AutoHeight** method:

```
iGrid1.Footer.Rows[1].AutoHeight();
```

10.3. Footer Cells

The footer section contains cells that are very similar to normal grid cells. Footer cells implement almost all features of normal grid cells, including cell merging and tooltips for truncated cell texts:



The two key differences between normal cells and footer cells are as follows:

- Footer cells cannot be edited by the user.
- Footer cells can display summaries calculated automatically.

To access footer cells and change their properties, use the **Cells** collection of the **Footer** object property of iGrid. Footer rows are indexed from top to bottom, and the top row always has the index of 0. Thus, if you want to display the value of 100 in the footer cell for the 5th column in the top (or only) footer row, you need code like this:

```
iGrid1.Footer.Cells[0, 4].Value = 100;
```

Every footer cell is represented with an instance of the **iGFooterCell** class. It implements almost all properties of normal cells (i.e. the **iGCell** class) — **Value**, **ImageIndex**, **BackColor**/**ForeColor**, **Style**, **TextTrimming**, and so on. Footer cells also support styles, which are objects of the **iGFooterCellStyle** class.

One of the main ideas that works for an iGrid footer cell is the inheritance of the column cell style (the **CellStyle** property of **iGCol**). If you do not redefine formatting properties of a footer cell, it inherits the look of the cells in the same column (the same format string, alignment, etc.) You can redefine any of these properties if required — for instance, highlight special values with a bold font and/or the red color.

Fine tuning of footer cell contents and look can be done dynamically with the **FooterCellDynamicContents**, **FooterCellDynamicFormatting**, **FooterCellDynamicStringFormat** events of iGrid. They work like their counterparts for normal cells (**CellDynamicContents**, **CellDynamicFormatting**, and **CellDynamicStringFormat**). For example, the following event handler will display all negative total values in the footer in red:

```
private void iGrid1_FooterCellDynamicFormatting(
  object sender, iGFooterCellDynamicFormattingEventArgs e)
{
  if (e.Total < 0)
    e.ForeColor = Color.Red;
}
```

Some additional points related to iGrid footer cells are as follows:

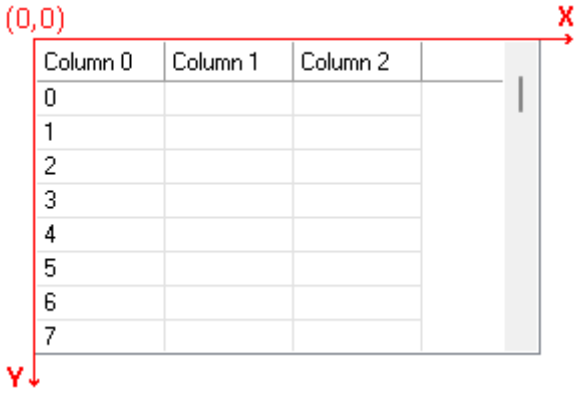
- Footer cells support the classic set of mouse events: **FooterCellMouseDown**, **FooterCellMouseMove**, **FooterCellMouseUp**, **FooterCellMouseEnter**, **FooterCellMouseLeave**. They are raised not only for normal footer cells, but also for the footer row header cell and the footer extra cell.
- Footer cells can be merged using the **SpanRows** and **SpanCols** properties of the **iGFooterCell** object (the same functionality is used for column headers).
- Footer cells have built-in tooltips, which can be redefined on the fly using the **RequestFooterCellToolTipText** event like we can do that for normal cells or column headers.
- Footer cells support custom drawing like normal cells (see the **CustomDrawFooterCellBackground** and **CustomDrawFooterCellForeground** events of **iGrid**). The row header area in the footer section also supports custom drawing — the **CustomDrawFooterRowHdr** event allows you to draw your custom contents over the standard color fill.

The functionality of footer summaries is described in greater detail in the Footer Summaries sub-section in this manual.

11. COORDINATES OF IGRID ELEMENTS

11.1. iGrid Coordinate System

iGrid uses a standard two-dimensional coordinate system for all spatial operations, including cell positioning, mouse event handling, and boundary calculations. All coordinates in iGrid are measured in pixels and are relative to the upper-left corner of the grid control, which serves as the origin point (0, 0). The x-axis is a horizontal axis increasing from left to right. The y-axis is a vertical axis increasing from top to bottom. The origin point is always located in the top-left corner of the control — even if the right-to-left mode is on. If iGrid has a border, its top-left corner coincides with the coordinate origin:



All main UI elements of iGrid are rectangles. The position of a UI element inside the control is determined by the x and y coordinates of the top-left corner and the width and height parameters. You can retrieve these values, and you can also determine the iGrid element at the specified coordinates using iGrid members. For more details, see the Retrieving Coordinates of iGrid Elements and Obtaining iGrid Element at Specified Coordinates topics.

As is customary in WinForms, coordinates of iGrid elements are always measured in pixels. And you can use the coordinates passed in event arguments of common WinForms events (such as **MouseDown**) in iGrid members as well.

If iGrid has scroll bars and the user is scrolling the contents of the control, the coordinates of moveable elements are changed respectively. For example, the y-coordinate of the top edge of the 10th row may equal 175 when iGrid is displayed on the screen for the first time. If the user clicked the down arrow button on the vertical scroll bar to scroll the iGrid contents up a bit, the y-coordinate of the top edge of the 10th row changes downward — for example, to 159 — and so on.

11.2. Retrieving Coordinates of iGrid Elements

You can retrieve the x-coordinate of the left edge of a column with the **X** property of the corresponding **iGCol** object representing the column, for example:

```
int mySecondColX = iGrid1.Cols[1].X;
```

The **Width** property of the **iGCol** object returns the width of the column.

Similar to columns, you can retrieve the y-coordinate of the top edge of a row of with the **Y** property of the corresponding **iGRow** object representing the row. The **Height** property of the **iGRow** object returns the height of the row.

To retrieve the coordinates of a cell, use the **Bounds** property or the **GetBounds** method of the **iGCell** class. The **Bounds** property returns the bounds of the area in which iGrid draws the cell's contents, while the **GetBounds** method can take into account row level indent if the cell is in the first column. If the cell is located in the first column and the row has a level indent, the **Bounds**

property returns the bounds of the cell excluding the indent. To get the entire bounds of the cell including the indent, use the **GetBounds** method specifying True to its parameter.

The **iGCell** class provides you with the **TextBounds** property allowing you to determine the area in which iGrid draws the text of the cell. This property can be useful if you want to show a custom edit control in a cell. A similar **iGCell**.**ImageBounds** property returns the area occupied by the cell image (if any). You can use it, for example, to track clicks exactly on cell images.

Like for normal cells, you can retrieve the coordinates of a footer cell with the **Bounds** property of the corresponding **iGFooterCell** object representing the footer cell. The **TextBounds** and **ImageBounds** properties of the **iGFooterCell** object return the rectangles occupied by the footer cell text and image (if any).

The coordinates of a column header can be retrieved with the **Bounds** property of the **iGColHdr** class. This property also returns the correct column header coordinates when the column header is in the group box.

The coordinates of the row header for a particular row are returned by the **HdrBounds** property of the corresponding **iGRow** object.

11.3. Obtaining iGrid Element at Specified Coordinates

iGrid members allow you to determine not only the coordinates of its elements on the screen, but also to solve the inverse problem — to find out what object is at a given point. You can do this for columns, rows, cells, column headers and footer cells.

The **FromX** method of the **Cols** collection is used to determine the column containing a given x-coordinate, for example:

```
iGCol myColInTheMiddle = iGrid1.Cols.FromX(iGrid1.Width / 2);
```

If there is no column containing the specified x-coordinate, **FromX** returns null.

Similar to columns, the **FromY** method of the **Rows** collection is used to determine the row containing a given y-coordinate.

To determine the cell containing a given point, use the **FromPoint** method of the **Cells** collection of iGrid. This method returns an empty object reference (null in C# and Nothing in VB) if there is no cell at the specified point. This feature can be used to implement the ability to deselect cells when the user clicks in the empty space of the cells area, for example:

```
private void iGrid1_MouseUp(object sender, MouseEventArgs e)
{
  if (iGrid1.Cells.FromPoint(e.X, e.Y) == null)
    iGrid1.PerformAction(iGActions.DeselectAllCells);
}
```

One of the overloaded versions of the method allows you to take into account the row level indent area when you determine the cell at the specified point. Note that this method can also be used for row text cells.

To determine the column header containing a given point, use the **FromPoint** method of the **Cells** collection of the **Header** object property of iGrid. Note that this function can also find a column header when it is inside the group box.

There is a similar method for the footer area allowing you to determine the footer cell at the specified coordinates. This is the **FromPoint** method of the **Cells** collection of the **Footer** object property of iGrid.

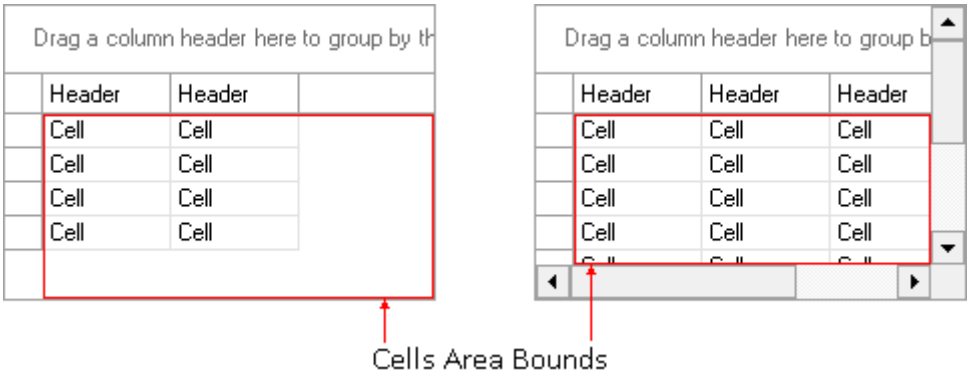
12. VIEWPORT

12.1. Viewport and Cells Area

Viewport in computer interfaces is an area in which the user can see and interact with the contents of an object. In the case of iGrid, this is the area inside the control in which the user sees iGrid elements representing data — cells, column headers, row headers, footer cells. The border of the control and its scroll bars are not included into the viewport.

The main part of the iGrid viewport is its cells area. It is a part of iGrid in which normal cells are drawn. iGrid allows you to retrieve the parameters of this area and set some of them.

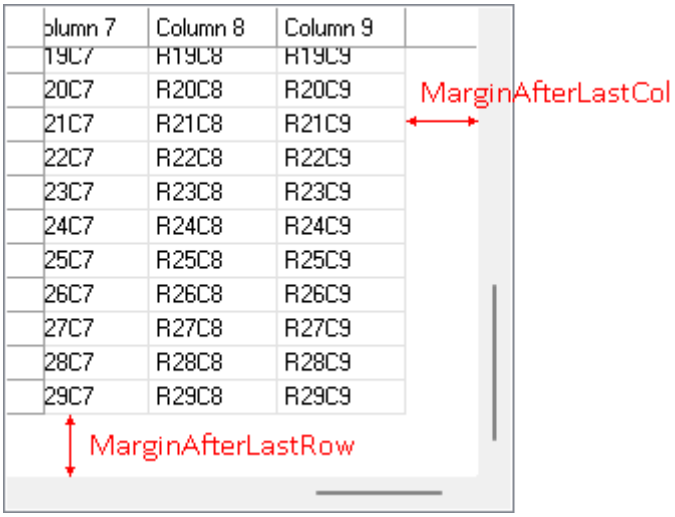
The **CellsAreaBounds** property of iGrid allows you to determine the coordinates of the rectangle in which iGrid cells are drawn. The rectangle returned by this property does not depend on the count and size of the rows and columns because this is the area intended for drawing cells. This rectangle is the whole area of iGrid excluding its border, group box, header, footer, and scroll bars:



A related member of iGrid is the **GetStartEndCells** method allowing you to know the top-left and bottom-right cells currently displayed in the viewport. Knowing these cells, you can determine the index of the first/last column/row currently drawn on the screen (in the viewport). Below is an example demonstrating how to display the index of the first and last grid rows visible in the viewport in a **Label** control:

```
iGCell myStart, myEnd;
Rectangle myStartBounds, myEndBounds;
iGrid1.GetStartEndCells(
  out myStart, out myStartBounds, out myEnd, out myEndBounds);

labelInfo.Text = string.Format(
  "First row index is {0}, last row index is {1}",
  myStart.RowIndex, myEnd.RowIndex);
```

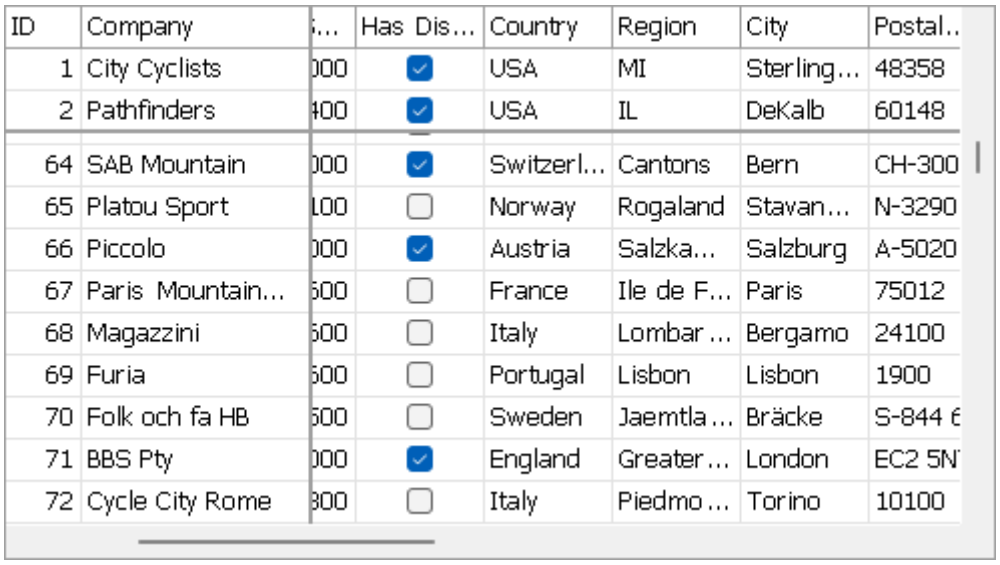
If iGrid has the vertical scroll bar and you scroll the grid to the last position in the vertical direction, by default there is no empty space between the bottom edge of the last row and the bottom edge of the viewport. The same is true for the horizontal scroll bar and the horizontal scrolling of the iGrid contents. The two special iGrid properties, **MarginAfterLastCol** and **MarginAfterLastRow**, allow you to specify the size of the empty space after the last column and row respectively:



These properties allow you to add some empty space to better separates the cells from other parts of iGrid when the grid contents have been scrolled to the last horizontal or vertical position. For example, the **MarginAfterLastRow** property is helpful if you want to add some space between the last row and the footer section if it is visible, or when you are implementing integral scrolling by rows (see the Special Features of Scroll Bars topic).

12.2. Freezing Rows and Columns

iGrid allows you to make some first columns and rows non-scrollable ("freeze" them) so that they will be always visible in the viewport. The grid on the screenshot below has 2 frozen rows and columns separated from scrollable rows and columns with 2-pixel lines of a dark gray color:



The **FrozenArea** object property of iGrid is used to set up the frozen columns and rows area. The two main properties of the **iGrid.FrozenArea** object are **ColCount** and **RowCount**. They are used to specify the numbers of columns and rows to freeze respectively. For example, you can freeze the first two columns using this statement:

```
iGrid1.FrozenArea.ColCount = 2;
```

The object returned by the **FrozenArea** property of iGrid provides you with the following properties you can use to adjust the frozen area:

PROPERTY	DESCRIPTION
ColCount	Get or sets the total number of the non-scrollable columns.
ColsEdge	Gets or sets the style of the grid line which separates the frozen columns from non-frozen ones. If the width of this line is set to zero, then the frozen columns will be separated by the normal vertical grid line used in other cells (see the **Vertical** property of the **GridLines** object property of the iGrid class).
RowCount	Get or sets the total number of the non-scrollable rows.
RowsEdge	Gets or sets the style of the grid line which separates the frozen rows from non-frozen ones. If the width of this line is set to zero, then the frozen rows will be separated by the normal horizontal grid line used in other cells (see the **Horizontal** property of the **GridLines** object property of the iGrid class).
SortFrozenRows	Gets of sets the value indicating whether to sort the frozen rows.

Note that row levels do not work in the frozen rows. The rows in the frozen area are always drawn as if they were rows without level indents.

12.3. Ensuring a Cell/Column/Row is Visible in the Viewport

When you need to make a specific cell visible in the viewport from code, call the cell's **EnsureVisible** method. If the viewport is too small to display all rows or columns, iGrid automatically scrolls its contents so the cell appears in view. For example:

```
iGrid1.Cells[49, 3].EnsureVisible();
```

To make a specific column or row visible in the viewport, call the corresponding **EnsureVisible()** method of the **iGCol** or **iGRow** object representing a column and row respectively.

The **EnsureVisible()** methods described above scroll the grid so that the target object is shown, preferably near the middle of the viewport. If you need to place it at an exact position, use the **Value** property of the corresponding **HScrollBar** or **VScrollBar** object property of iGrid to scroll the contents to the required position. For example, if all rows have the default height, the following statement places the 10th row directly below the header area:

```
iGrid1.VScrollBar.Value = iGrid1.DefaultRow.Height * 9;
```

If rows have different heights, calculate the scroll bar value from the total height of all preceding rows:

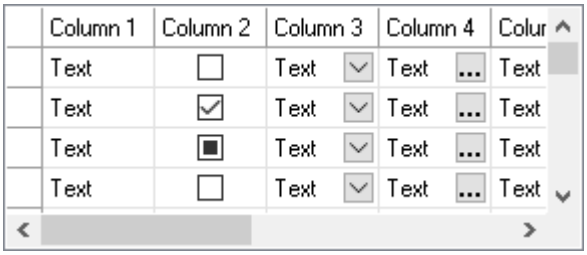
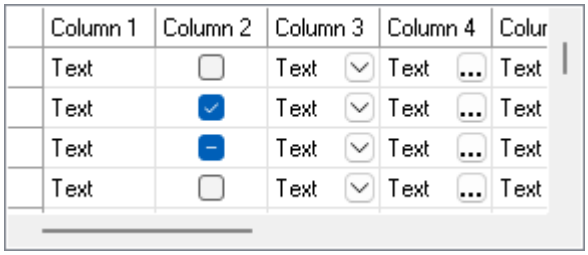
```
int myScrollValue = 0;

for (int i = 0; i < 10; i++)
  myScrollValue += iGrid1.Rows[i].Height;

iGrid1.VScrollBar.Value = myScrollValue;
```

13. RENDERING STYLES

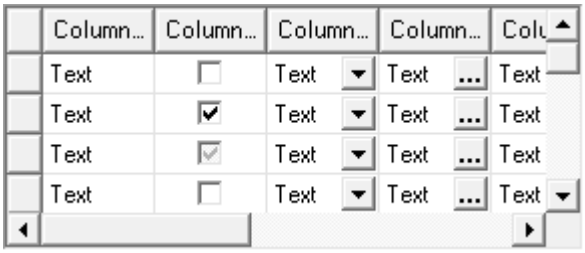
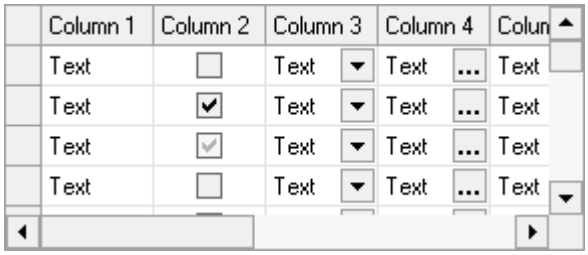
13.1. Built-in Rendering Styles

iGrid allows you to select one of the three predefined rendering styles — the system style, the classic 3D style, and the flat style. You can specify the desired rendering style for the entire control by using the **RenderStyle** property of iGrid. It accepts a value from the **iGRenderStyle** enumeration. The **System**, **Classic**, and **Flat** enumeration values correspond to the three rendering styles listed above.

One of the key ideas behind iGrid is to match the look and feel of the operating system as closely as possible, and the system rendering style is used by default for that purpose. This style corresponds to the **System** value of the **RenderStyle** property. The following screenshots show an iGrid control rendered using the system style in Windows 10 and Windows 11. The screenshots illustrate that iGrid truly follows the operating system's visual style and therefore appears differently in different versions of Windows:

WINDOWS 10	**WINDOWS 11**
	

If visual styles are unavailable for any reason, such as being disabled in the operating system, or if **RenderStyle** is set to **Classic**, iGrid uses the classic 3D look. The third value from the **iGRenderStyle** enumeration, **Flat**, allows you to make iGrid flat:

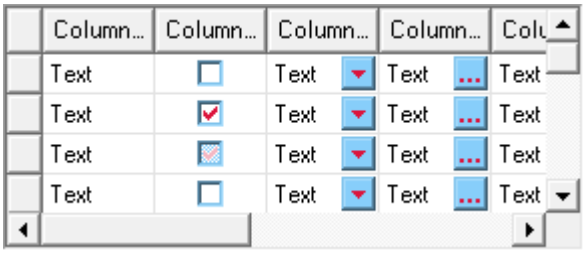
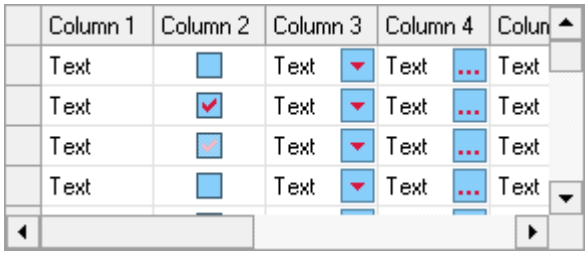
3D STYLE	**FLAT STYLE**
	

When the system rendering style is used, column and row headers are drawn as flat rectangles whose borders are formed by the grid lines on their right and bottom sides. By default, these elements use colors that match the current system colors as closely as possible, although you can customize these colors if necessary using iGrid properties. Cell controls, such as combo buttons and check boxes, as well as scroll bars, are drawn by the operating system using its current visual style, and their colors therefore cannot be changed.

In the classic and flat styles, cell controls and scroll bars are drawn entirely by iGrid, allowing you to adjust their colors if needed. For cell controls, this can be done with the help of the **CellCtrlBackColor** and **CellCtrlForeColor** properties of iGrid. For example, the following settings

```
iGrid1.CellCtrlBackColor = Color.LightSkyBlue;
iGrid1.CellCtrlForeColor = Color.Crimson;
```

will have the following effect in the classic and flat styles:

3D STYLE	FLAT STYLE
	

You can learn more about the color customization options for other areas of iGrid in the following topics:

- Header Section Drawing Settings
- Row Header Area Drawing Settings
- Scroll Bar Properties

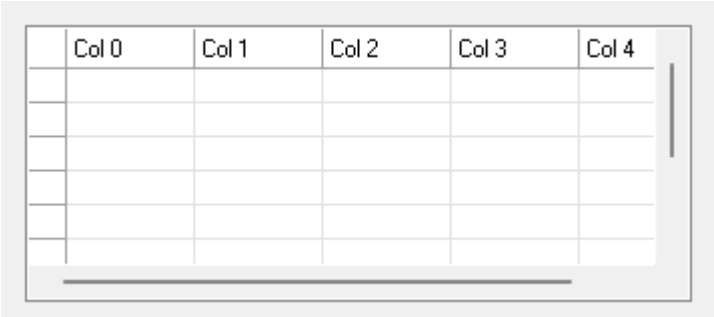
The iGrid border is a special control element that can use a rendering style different from the one specified by the **RenderStyle** property. By default, it is rendered using the same style as the rest of the control. The **BorderStyle** property of iGrid lets you select a different style or remove the border entirely. Read the Border Styles and Settings topic to learn more about customizing the border.

13.2. Border Styles and Settings

The iGrid border can be rendered with one of the built-in styles of the iGrid control (see the Built-in Rendering Styles topic for more information). These are the system border, the classic 3D border, and the flat border. If required, you can also remove the border around iGrid entirely.

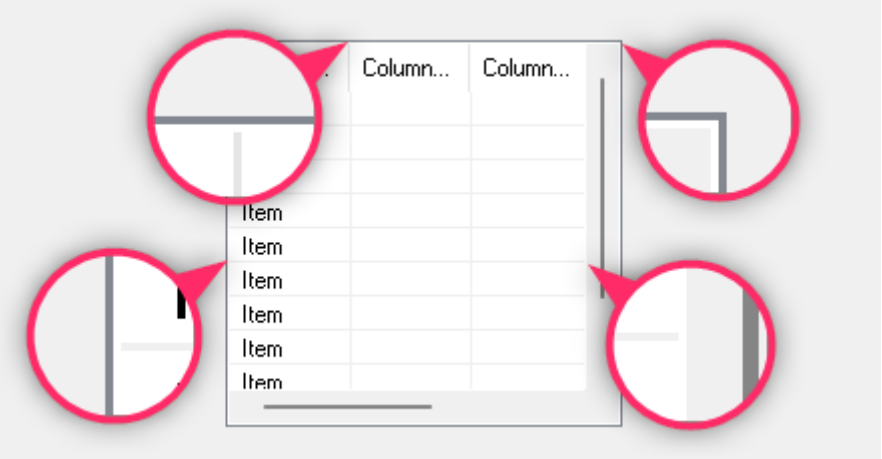
The **BorderStyle** property of iGrid is used to select one of these predefined border styles or remove the border. This property accepts the following values from the **iGBorderStyle** enumeration: **Auto**, **System**, **Classic**, **Flat**, or **None**. The default value is **Auto**, which allows the border to inherit the style specified by the **RenderStyle** property of iGrid.

If iGrid uses the system style and the border inherits this style, or if the **BorderStyle** property is set to **System**, iGrid draws a system border similar to the one used by other WinForms list controls, such as **ListBox** and **ListView**. The screenshot below shows how this border looks in iGrid on Windows 11:



The look of the default system border depends on the availability of the OS visual styles. iGrid renders its border using the OS visual styles if they are available; if not, iGrid draws the classic 3D border. This behavior fully corresponds to the behavior of the WinForms list controls when their **BorderStyle** property is set to the default **Fixed3D** value.

If you look closely at the system border rendered with OS visual styles, you will see that it consists of two lines: a 1-pixel dark outer line and a 1-pixel white inner line. This is how the OS renders borders in native list controls such as ListView:



Notice that the same 1-pixel white line is drawn at the left of the vertical scroll bar and at the top of the horizontal scroll bar, which creates a beautiful symmetric effect of empty space around the scroll bars. iGrid uses all these effects if its constituent parts including border are rendered with the OS visual styles (the default rendering style). If the visual styles are not available and iGrid automatically switches to the 3D border, its width also equals 2 pixels. This guarantees that the size of the client area of the control always remains unchanged regardless of the border look and the user will always see the same contents inside iGrid.

If you need the classic 3D border that does not depend on the availability or look of the OS visual styles, set the **BorderStyle** property of iGrid to **Classic**. The classic 3D border is drawn with the corresponding system colors.

An alternative way to provide a border that will always look the same is to use a flat border. If you assign the **Flat** value to the **BorderStyle** property of iGrid, you get a 1-pixel solid flat border around the iGrid contents. In this case two other properties of iGrid, **BorderColor** and **BorderWidth**, can be used to change the default color and thickness of the flat border. The following example shows how to make the border of an iGrid control a thin dark blue line:

```
iGrid1.BorderStyle = iGBorderStyle.Flat;
iGrid1.BorderColor = Color.DarkBlue;
```

In some scenarios you may need to know the effective width of the iGrid border for coordinate-related calculations. It may be a tedious task to take into account all factors affecting the iGrid border and code this task yourself. The **GetEffectiveBorderWidth** method of iGrid can be used to retrieve the effective border width with one statement.

13.3. Custom Rendering Styles

The concept

iGrid allows you to redefine the drawing of its constituent parts — column headers, scroll bars, ellipsis and combo buttons in cells, etc. Thus, you can apply your own visual style to iGrid if you do not like the standard drawing provided by the control. Another case when this feature is useful is when you need to incorporate iGrid into a form that uses a non-standard style and iGrid should have the same look and feel.

To redefine the drawing of iGrid parts, first define a class implementing the **IiGControlPaint** interface and implement the custom drawing of the corresponding parts in its members. Then create an instance of this class and assign it to the **CustomControlPaint** property of iGrid.

You may redefine the drawing of all parts of iGrid or only some of them. The value returned by the **SupportedFunctions** property of the object implementing the **IiGControlPaint** interface tells iGrid which parts are redefined. This value is a combination of the flags defined in the **iGControlPaintFunctions** enumeration. The drawing for these parts is implemented in the

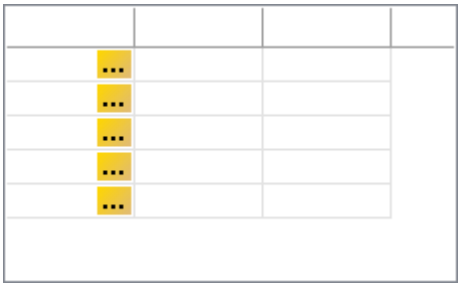
corresponding members of the **IiGControlPaint** interface. They are called by iGrid when it draws the corresponding parts.

If you redefine the drawing not for all parts, you implement only the members that will be used — the rest members may remain non-implemented. The help topic for the **IiGControlPaint.SupportedFunctions** property contains a table in which you can see which members of the **IiGControlPaint** interface must be implemented for every **iGControlPaintFunctions** flag.

Note that an object implementing **IiGControlPaint** can be shared with all iGrid instances in your application. This allows you to create just one instance of this custom control painter in the whole application.

An example

To demonstrate how this approach works in practice, let's consider an example in which we redefined the drawing of ellipsis buttons in cells. In this example, we used a yellow linear gradient as their background, as shown in the screenshot below:



To create ellipsis buttons like these, we first defined a class that implements our custom drawing logic:

```
public class CustomEllipsisButtonRenderer: IiGControlPaint
{
  iGControlPaintFunctions IiGControlPaint.SupportedFunctions
  {
    get { return iGControlPaintFunctions.EllipsisButton; }
  }

  void IiGControlPaint.DrawEllipsisButton(
    iGrid grid, Graphics g,
    int x, int y, int width, int height,
    iGControlState controlState, iGColorMode colorMode)
  {
    using (LinearGradientBrush myBrush = new LinearGradientBrush(
      new Point(x, y),
      new Point(x + width + 1, y + height + 1),
      Color.Gold,
      Color.BurlyWood))
    {
      g.FillRectangle(myBrush, x, y, width, height);
    }
  }

  // Other non-implemented IiGControlPaint members (stubs)
  ...
}
```

Next, we created an instance of this class and assigned it to the **CustomControlPaint** property of iGrid. The iGrid shown in the screenshot above was created using the following code:

```
iGrid1.Cols.Count = 3;
iGrid1.Cols[0].CellStyle.TypeFlags = iGCellTypeFlags.HasEllipsisButton;

iGrid1.Rows.Count = 5;

iGrid1.CustomControlPaint = new CustomEllipsisButtonRenderer();
```

14. GRID LINES

14.1. Cell Grid Lines

Grid lines are special lines that form part of the grid's visual appearance. Vertical grid lines are drawn along the right edges of columns, while horizontal grid lines are drawn along the bottom edges of rows. Each grid line belongs to the corresponding column or row and is included in its width or height.

iGrid allows you to control the visibility of these cell grid lines and configure their style properties, such as width, color, and dash style. Most properties for configuring cell grid lines are grouped in the **GridLines** object property of iGrid.

The following properties of the **GridLines** object specify the styles of the cell grid lines:

PROPERTY	DESCRIPTION
Horizontal	Gets or sets the style of the horizontal grid lines.
HorizontalLastRow	Gets or sets the style of the horizontal grid line in the last visible row.
HorizontalExtended	Gets or sets the style of the extended horizontal grid lines.
Vertical	Gets or sets the style of the vertical grid lines.
VerticalLastCol	Gets or sets the style of the vertical grid line in the last visible column.
VerticalExtended	Gets or sets the style of the extended vertical grid lines.

A grid line style is defined by an **iGPenStyle** object. Its **Color**, **DashStyle**, and **Width** properties specify the color, dash style, and width of the grid line, respectively.

By default, the **Color** property of the grid line style object for every grid line is set to **Color.Empty**. This means that the color corresponding to the current iGrid theme is used automatically. The **DashStyle** and **Width** properties are set to **DashStyle.Solid** and 1 by default.

The **Mode** property of the **GridLines** object determines which grid lines are displayed: horizontal lines, vertical lines, or both. You can display both horizontal and vertical grid lines, which is the default, display only horizontal or only vertical grid lines, or hide grid lines altogether.

When a grid line is hidden through the **Mode** property, it becomes invisible but still occupies space in the cell. To remove a grid line completely and free the space it occupies for the cell contents, set the **Width** property of its style to 0.

The **VerticalLastCol** and **HorizontalLastRow** properties allow you to set special styles for the lines framing the cell area. Take this into account if you want to redefine all vertical and/or horizontal grid lines for cells. For example, if you want to use the same color for all vertical grid lines, you should specify it in both **Vertical** and **VerticalLastCol** properties of the **GridLines** object:

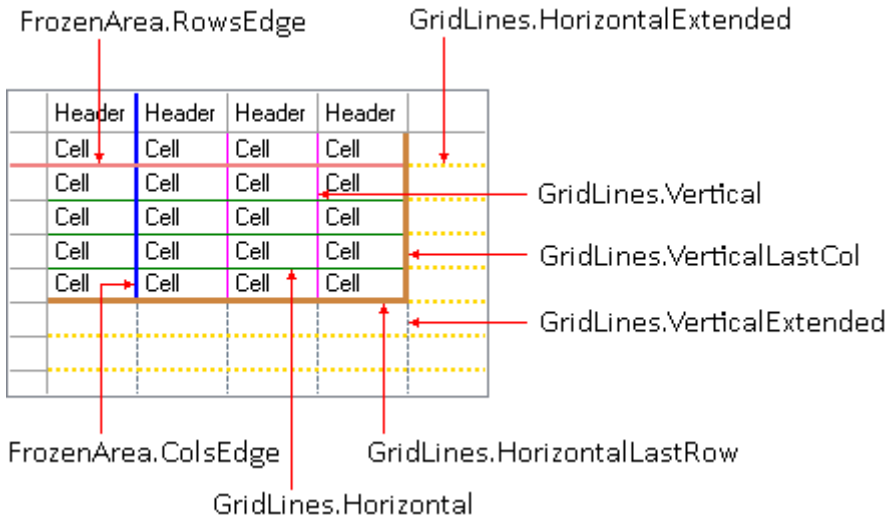
```
iGrid1.GridLines.Vertical.Color = Color.OrangeRed;
iGrid1.GridLines.VerticalLastCol.Color = Color.OrangeRed;
```

The same is true for the horizontal grid lines (**GridLines**.**Horizontal**, **GridLines**.**HorizontalLastRow**).

The **ExtendMode** property of the **GridLines** object allows you to specify whether to extend grid lines horizontally and vertically over the cell-free area. To adjust the styles of the extended grid lines, use the **HorizontalExtended** and **VerticalExtended** properties. You can see the effect of these properties when the cells do not occupy the whole client area.

The edge lines of the frozen area can also be adjusted independently from all other grid lines. This is done with the help of the **ColsEdge** and **RowsEdge** properties of the **FrozenArea** object property of iGrid.

The following picture and accompanying code snippet show how you can configure cell grid lines in iGrid:

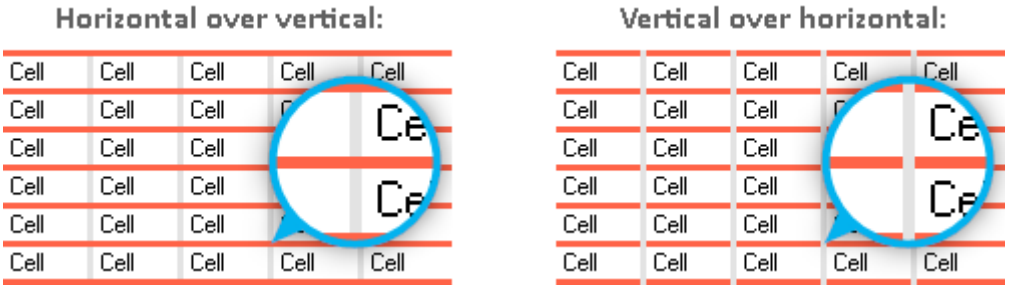



```
iGrid1.RowHeader.Visible = true;

iGrid1.DefaultCol.Text = "Header";
iGrid1.DefaultCol.Width = 45;
iGrid1.DefaultCol.DefaultCellValue = "Cell";

iGrid1.Cols.Count = 4;
iGrid1.Rows.Count = 5;

iGrid1.FrozenArea.ColCount = 1;
iGrid1.FrozenArea.RowCount = 1;

iGrid1.FrozenArea.ColsEdge.Color = Color.Blue;
iGrid1.FrozenArea.RowsEdge.Color = Color.LightCoral;

iGrid1.GridLines.Horizontal.Color = Color.Green;
iGrid1.GridLines.Vertical.Color = Color.Magenta;

iGPenStyle myPenStyle = new iGPenStyle(
  Color.Peru, 3, System.Drawing.Drawing2D.DashStyle.Solid);
iGrid1.GridLines.HorizontalLastRow = myPenStyle;
iGrid1.GridLines.VerticalLastCol = myPenStyle;

iGrid1.GridLines.ExtendMode = iGGridLinesMode.Both;
iGrid1.GridLines.HorizontalExtended = new iGPenStyle(
  Color.Gold, 2, System.Drawing.Drawing2D.DashStyle.Dot);
iGrid1.GridLines.VerticalExtended.Color = Color.SlateGray;
iGrid1.GridLines.VerticalExtended.DashStyle =
  System.Drawing.Drawing2D.DashStyle.Dash;
```

Note that if the frozen areas are enabled, their edge lines are extended to the header areas automatically to frame non-scrolled areas in the header areas. However, this does not happen for the special grid lines framing the last visible columns and rows. If iGrid did that, the user could see a strange picture in iGrid with the default properties because the last column header would use a lighter vertical grid line in contrast to the dark vertical grid line in other column headers (the same is true for the last horizontal grid line in row headers).

14.2. Z-Order of Cell Grid Lines

If you use vertical and horizontal grid lines of the same color, there is no difference whether vertical grid lines are drawn over horizontal grid lines or vice versa. But if their colors differ, the order in which the vertical and horizontal grid lines are drawn yields different effects:



iGrid allows you to specify the z-order of the vertical and horizontal cell grid lines to achieve the desired visual effect like on the above pictures. This is done with the **GridLines**.**ZOrder** property. This property accepts one of the two values from the **iGGridLinesZOrder** enumeration — **HorizontalOverVertical** or **VerticalOverHorizontal**. The default is **HorizontalOverVertical**.

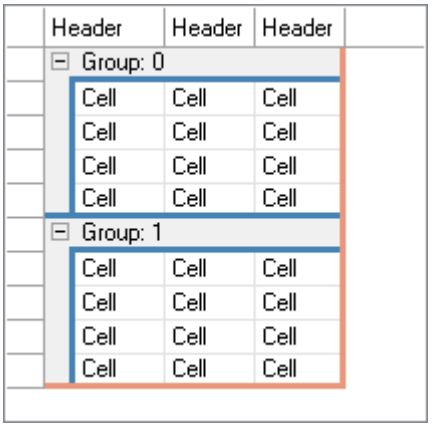
As you already know from the Cell Grid Lines topic, iGrid cell grid lines fall into one of the following categories:

1. Normal grid lines (the **Horizontal** and **Vertical** properties of the **GridLines** object property).
2. Grid lines bounding frozen areas (the **ColsEdge** and **RowsEdge** properties of the **FrozenArea** object property).
3. Grid lines bounding the whole cell area (the **HorizontalLastRow** and **VerticalLastCol** properties of the **GridLines** object property).

iGrid is based on the concept of drawing priority for grid lines of different kinds. Every kind of grid lines is drawn in its virtual plane. The lower the priority of a grid line, the lower plane it is located in. The normal cell grid lines have the lowest priority, then the frozen edges go, and the lines bounding the whole cell area have the highest priority. As a result, grid lines of one kind cannot intersect grid lines of another kind. This system guarantees that (1) normal cell grid lines never break frozen area edges and (2) the grid lines bounding the whole cell area always look solid because they cannot be broken by normal cell grid lines or frozen area edges. Actually the aforementioned **ZOrder** property of the **GridLines** object property specifies the z-order of vertical and horizontal grid lines in the plane of every kind of grid lines.

14.3. Group Row Grid Lines

The style of the horizontal grid lines in group rows is set with the **GroupRows** property of the **GridLines** object. Below is an example of a thick blue grid line for group rows set with this property:



The corresponding code snippet that sets up these grid lines looks as follows:

```
iGrid1.GridLines.GroupRows.Color = Color.SteelBlue;
iGrid1.GridLines.GroupRows.Width = 3;

iGPenStyle myPenStyle = new iGPenStyle(Color.DarkSalmon, 3,
System.Drawing.Drawing2D.DashStyle.Solid);
iGrid1.GridLines.HorizontalLastRow = myPenStyle;
iGrid1.GridLines.VerticalLastCol = myPenStyle;
```

Note that the vertical line in the indent area has the same style as the group row grid line.

14.4. Column Header and Row Header Grid Lines

iGrid column headers and row headers can also be considered as cells with their own grid lines. If the header areas are not rendered with the special relief 3D style, it is possible to adjust the grid lines for column and row headers the same way as for normal cells.

The **HGridLinesStyle**, **VGridLinesStyle** and **SeparatingLine** properties of the **Header** object property are used to configure the grid lines in the header area. The **HGridLinesStyle** and **VGridLinesStyle** properties of the **RowHeader** object are used to configure the grid lines for row headers. You can find more information about column header and row header grid lines in the Header Section Drawing Settings and Row Header Area Drawing Settings topics.

15. SCROLL BARS

15.1. Scroll Bar Properties

iGrid allows you to adjust the appearance and behavior of its scroll bars to suit your needs. It is done with the **ScrollBarSettings**, **HScrollBar**, and **VScrollBar** properties of iGrid.

The **ScrollBarSettings** object property defines the general look of iGrid scroll bars with its properties:

PROPERTY	DESCRIPTION
BackColor	Gets or sets the background color of the scroll bars. The value of this property is ignored if the grid uses the OS visual styles.
ForeColor	Gets or sets the color of the scroll bar arrows. The value of this property is ignored if the grid uses the OS visual styles.
ImageList	Gets or sets the image list used by the custom buttons.
Opacity	Gets or sets a numeric value indicating the opacity of the scroll bars. The value of this property has the **Double** data type and can vary from 0 (invisible) to 1 (opaque).

The **RenderStyle** property of iGrid defines the look of iGrid including the style of its scroll bars. You can find out more about the built-in rendering styles in iGrid from the Built-in Rendering Styles topic.

The **HScrollBar** and **VScrollBar** properties return instances of the **iGScrollBar** class representing each scroll bar. The following table lists the properties of this class you can use to adjust a scroll bar:

PROPERTY	DESCRIPTION
CustomButtons	Represents the collection of the custom buttons of the scroll bar.
Enabled	Gets or sets a value indicating whether the scroll bar is enabled.
Height	Gets the height of the scroll bar.
HeightOverride	Allows you to set custom height for the horizontal scroll bar.
LargeChange	Gets the value to be added to or subtracted from the **Value** property when the scroll box is moved a large distance. This value is actually the height (or the width for horizontal scroll bar) of the scrollable client area which corresponds to one visible page of cells on the screen.
Locked	Returns a value indicating whether the scroll bar is locked by iGrid (read-only). If you wish to lock the scroll bar in code, use the **Enabled** property.
Maximum	Gets the upper limit of values of the scrollable range.
Minimum	Gets the lower limit of values of the scrollable range.

PROPERTY	DESCRIPTION
SmallChange	Gets or sets the value to be added to or subtracted from the **Value** property when the scroll box is moved a small distance.
Value	Gets or sets the numeric value that represents the current position of the scroll box.
Visibility	Gets or sets a value indicating whether the scroll bar should be always visible, only when necessary (normal mode), or never. See the detailed description of this property in the Special Features of Scroll Bars topic.
Visible	Gets a value indicating whether the scroll bar is visible. This property does not depend on the visibility of the whole control and returns True if the scroll bar is present in the grid even if the whole control can be hidden.
Width	Gets the width of the scroll bar.
WidthOverride	Allows you to set custom width for the vertical scroll bar.

The most important property of an **iGScrollBar** object is **Value**. It allows you to read the position of the scroll box, or scroll the grid from code when you assign a new value to this property.

15.2. Scroll Bar Events

iGrid provides you with various events allowing you to track and adjust scrolling.

The **HScrollBarScroll** or **VScrollBarScroll** events are raised after the scroll box has been moved and before the scroll bar value is changed. The **Type** argument of these events indicates how the scroll box was moved. Note that these events are raised only when a grid is scrolled with a scroll bar.

The **HScrollBarValueChanged** and **VScrollBarValueChanged** events are raised after iGrid was scrolled and the value of a scroll bar has been changed. These events are raised in all cases when a grid is being scrolled: with its scroll bar interactively, if a cell outside of the viewport was selected and iGrid is scrolled automatically to show this cell in the viewport, when the position of the scroll box on the scroll bar is changed from code, etc.

The example below shows how to synchronize the contents of two grids. When one of the grids is scrolled vertically, the other will be scrolled too.

```
private void iGrid1_VScrollBarValueChanged(object sender, System.EventArgs
e)
{
  iGrid2.VScrollBar.Value = iGrid1.VScrollBar.Value;
}

private void iGrid2_VScrollBarValueChanged(object sender, System.EventArgs
e)
{
  iGrid1.VScrollBar.Value = iGrid2.VScrollBar.Value;
}
```

The following example gives you an idea how to show a tip window at the bottom-left corner of a grid when the vertical scroll bar is being scrolled with the scroll box. The tip window in the example below is a descendant of the WinForms **Form** class and contains a **Label** control named 'fLabel'. The tip window displays the row indexes of the first and last row visible in the iGrid viewport:

```
private void SetTipText()
{
  //Get the first and last visible cells
  iGCell myStart, myEnd;
  Rectangle myStartBounds, myEndBounds;
  iGrid1.GetStartEndCells(
    out myStart, out myStartBounds, out myEnd, out myEndBounds);

  //Set the tip text
  TipForm.fLabel.Text = string.Format(
    "First row index: {0}\r\nLast row index: {1}",
    myStart.RowIndex, myEnd.RowIndex);
}

private void iGrid1_VScrollBarScroll(
  object sender, iGScrollEventArgs e)
{
  //Hide the tip form if scrolling is finished
  if (e.Type == iGScrollEventType.EndScroll)
    TipForm.Hide();
  else if (e.Type == iGScrollEventType.ThumbTrack)
  {
    //Show the tip form if it is not visible
    if (!TipForm.Visible)
    {
      Point myPoint = new Point(
        iGrid1.Bounds.Left,
        iGrid1.Bounds.Bottom - TipForm.Height);
      TipForm.Location = PointToScreen(myPoint);
      SetTipText();
      TipForm.Show();
    }
  }
}

private void iGrid1_VScrollBarValueChanged(
  object sender, System.EventArgs e)
{
  if (TipForm.Visible)
  {
    //Set the text to the tip form's label
    SetTipText();
  }
}
```

You can find other useful events related to scroll bars in iGrid. Among them are:

- **HScrollBarValueChanging**, **VScrollBarValueChanging**. They allow you to correct the **Value** property of the corresponding scroll bar before it is changed. This can help you if you implement your own logic of scrolling. See an example in the Special Features of Scroll Bars topic.
- **HScrollBarVisibleChanged**, **VScrollBarVisibleChanged**. These events notifies you when the corresponding scroll bar becomes visible or hidden.
- **HScrollBarHeightChanged**, **VScrollBarWidthChanged**. These events informs you about a change of the thickness of the corresponding scroll bar (for example, when the user changes it in the system settings).

15.3. Special Features of Scroll Bars

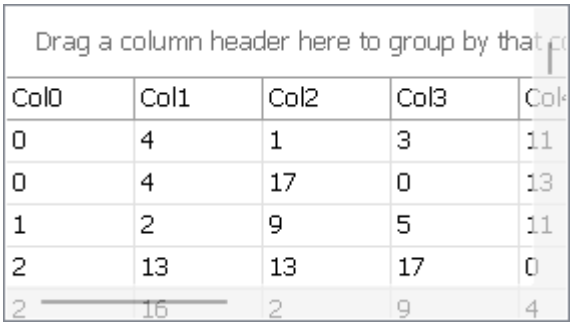
Visibility

Normally an iGrid scroll bar appears only if you do not see all the rows or columns in the viewable area and you need to scroll the grid's contents to see all the cells. But sometimes it is useful to never display a scroll bar or display it always even if all the cells are visible in the client area. iGrid provides you with the following 3 modes of scroll bar visibility which you can set through the **Visibility** property of the **VScrollBar** or **HScrollBar** object properties:

- **OnDemand**. This is the default mode. In this mode iGrid shows the scroll bars only when they are needed (not all the cells are in the visible area).
- **Always**. In this mode a scroll bar is always visible even if it is not needed. Use this mode if you place your custom buttons on the scroll bar and you want to give to the user the ability to access them at any time, even if all the cells are visible on the screen and there is no need for the scroll bar.
- **Hide**. In this mode a scroll bar is never visible even if not all the cells are entirely in the visible area. When this mode is turned on, you cannot scroll cells through visual interface. Use this mode if you want to show to the user only a limited part of the cells.

Opacity

iGrid supports semi-transparent scroll bars:



To make both scroll bars semi-transparent, use the **Opacity** property of the **ScrollBarSettings** object:

```
iGrid1.ScrollBarSettings.Opacity = 0.7;
```

This property of the **Double** data type enables you to specify the level of transparency for both scroll bars. The default value is 1.00 which means the scroll bars are opaque. When this property is set to a value less than 100% (1.00), the scroll bars are made more transparent. Setting this property to a value of 0% (0.00) makes the scroll bars completely invisible.

Small Change

The **SmallChange** property allows you to determine how many pixels to scroll when the user presses the scroll up/down or left/right buttons of the scroll bars. This value is equal to 16 for the vertical scroll bar and 10 for the horizontal scroll bar. However, you can change these values to adjust the scrolling in both directions for your needs.

Scrolling by rows instead of smooth scrolling

iGrid uses smooth scrolling for its rows and columns. This means you can scroll iGrid smoothly by pixels, but the first row or column in the viewport may be partially hidden by the corresponding cell area edge. To avoid this, you can adjust the scrolling behavior using the **VScrollBarValueChanging** and **HScrollBarValueChanging** events. They are raised when iGrid is about to assign a new value to the scroll bar **Value** property, and you can correct this value on-the-fly if required.

For instance, if you use rows of the same height, you can implement scrolling by rows using an event handler like this:

```
void iGrid1_VScrollBarValueChanging(
  object sender, iGScrollBarValueChangingEventArgs e)
{
   // The result of division of two integers is
   // always an integer in C#:
   e.Value = (e.Value / iGrid1.DefaultRow.Height) *
     iGrid1.DefaultRow.Height;
}
```

You also need to do one more adjustment to allow the user to see the last row without clipping in this scenario. Assign the value returned by the expression **iGrid1.DefaultRow.Height** to the **MarginAfterLastRow** property of iGrid for that.

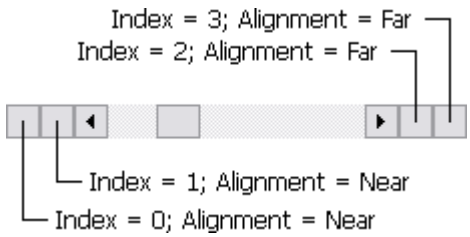
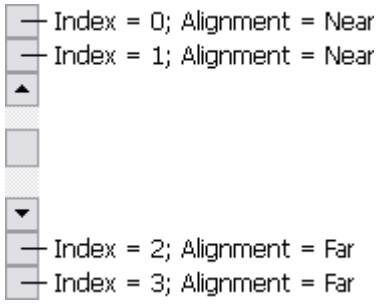
This technique is also fully applicable to the horizontal scrolling.

15.4. Custom Buttons in Scroll Bars

iGrid allows you to add custom buttons to its scroll bars. The **HScrollBar** and **VScrollBar** objects representing iGrid scroll bars provide you with the **CustomButtons** collections used for this purpose. Generally you use one of the overloaded versions of the **Add** method of this collection to create custom scroll bar buttons, for example:

```
iGrid1.HScrollBar.CustomButtons.Add(iGScrollBarCustomButtonAlign.Near);
iGrid1.VScrollBar.CustomButtons.Add(iGScrollBarCustomButtonAlign.Far);
```

Custom scroll bar buttons can be inserted into scroll bars before or after their standard parts, i.e. before or after arrow buttons. The alignment option of a custom button specifies its position:

HORIZONTAL SCROLL BAR	VERTICAL SCROLL BAR
	

Every custom scroll bar button is represented with an instance of the **iGScrollBarCustomButton** class. It implements the following properties:

PROPERTY	DESCRIPTION
Action	Gets or sets one the predefined standard actions to perform when the button is clicked. If the image for the action is allowed, it is also drawn in the button.
Alignment	Gets or sets the alignment of the scroll bar button. A scroll bar custom button can be aligned at the top or bottom in the vertical scroll bar, and on the left or right in the horizontal scroll bar.

PROPERTY	DESCRIPTION
Enabled	Gets or sets a value indicating whether the scroll bar button responds to user interaction.
ImageIndex	Gets or sets the index of the image displayed in the scroll bar button.
Tag	Gets or sets an object containing data about the scroll bar button.
ToolTipText	Gets or sets the tool tip text for the scroll bar button.

The default foreground of custom buttons on iGrid scroll bars is empty. To explain the purpose of a custom button to the user, you can display an icon or draw something on it using GDI+.

The source of icons for custom buttons is specified with the **ImageList** property of the **ScrollBarSettings** object. The image index for the button is specified in the **ImageIndex** property of the custom button.

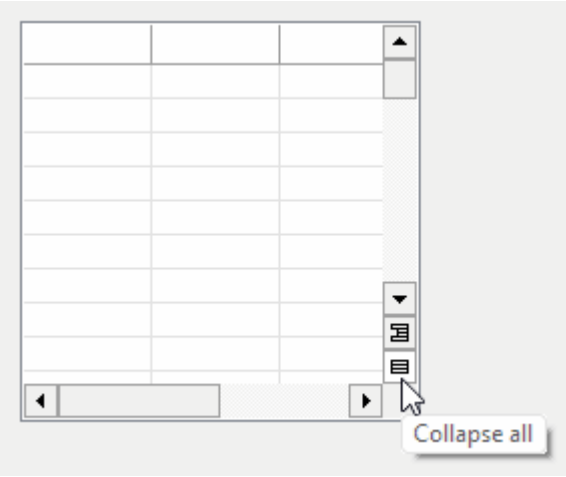
If you specify one of the possible user actions for the custom button with its **Action** property, iGrid will automatically draw the icon of the corresponding action if it is available. The list of available built-in icon actions can be found in the description of the **DrawActionGlyph** method.

In addition to that, you can also specify an optional tool tip text for the custom button with its **ToolTipText** property.

The overloaded versions of the **CustomButtons**.**Add** method allow you to specify alignment, icon, action, and tool tip text for a custom button in one call. Below is a classical example demonstrating how to add custom buttons for the frequently used 'Collapse All' and 'Expand All' actions to the vertical scroll bar:

```
iGrid1.VScrollBar.CustomButtons.Add(
  iGScrollBarCustomButtonAlign.Far,
  iGActions.ExpandAll, "Expand all");

iGrid1.VScrollBar.CustomButtons.Add(
  iGScrollBarCustomButtonAlign.Far,
  iGActions.CollapseAll, "Collapse all");
```

The result is on the screenshot below:



If you want to draw custom contents on a custom scroll bar button, use the **HScrollBarCustomButtonDrawBackground**, **VScrollBarCustomButtonDrawBackground**, **HScrollBarCustomButtonDrawForeground**, and **VScrollBarCustomButtonDrawForeground** events.

To process clicks on custom scroll bar buttons, handle the **HScrollBarCustomButtonClick** and **VScrollBarCustomButtonClick** events of iGrid. If a custom button has an associated user actions specified in its **Action** property, it will be performed automatically when this button is clicked and there is no need to write the corresponding event handler.

Custom buttons on scroll bars trigger other traditional mouse events you can use for your own needs: **HScrollBarCustomButtonMouseMove**, **VScrollBarCustomButtonMouseMove**, **HScrollBarCustomButtonMouseDown**, **VScrollBarCustomButtonMouseDown**, **HScrollBarCustomButtonMouseUp**, **VScrollBarCustomButtonMouseUp**, **HScrollBarCustomButtonMouseEnter**, **VScrollBarCustomButtonMouseEnter**, **HScrollBarCustomButtonMouseLeave**, **VScrollBarCustomButtonMouseLeave**.

One tip if you plan to use custom scroll bar buttons. By default, iGrid scroll bars appear only when they are needed, i.e. if there are cells outside of the grid viewport. If you want to make actions in custom scroll bar buttons available in any situation, make the corresponding scroll bar always visible by setting its **Visibility** property to **Always**.

16. ROW TEXTS

16.1. Row Text Column

iGrid rows can be group rows displaying one wide cell instead of normal row cells. Rows with normal cells can also display special wide cells below normal cells called "row text cells" (they are described in greater details in the Row Text Cells topic). The contents of group rows and row text cells are stored in the cells of a special column called "row text column".

The row text column is created automatically and cannot be deleted. It works like a normal column with some exceptions related to its nature. For example, iGrid does not display a column header for it in the header area containing column headers for all visible columns. The **Cols**.**Count** property always returns the total number of columns except the row text column, and the "for-each" construction never enumerates this column as well.

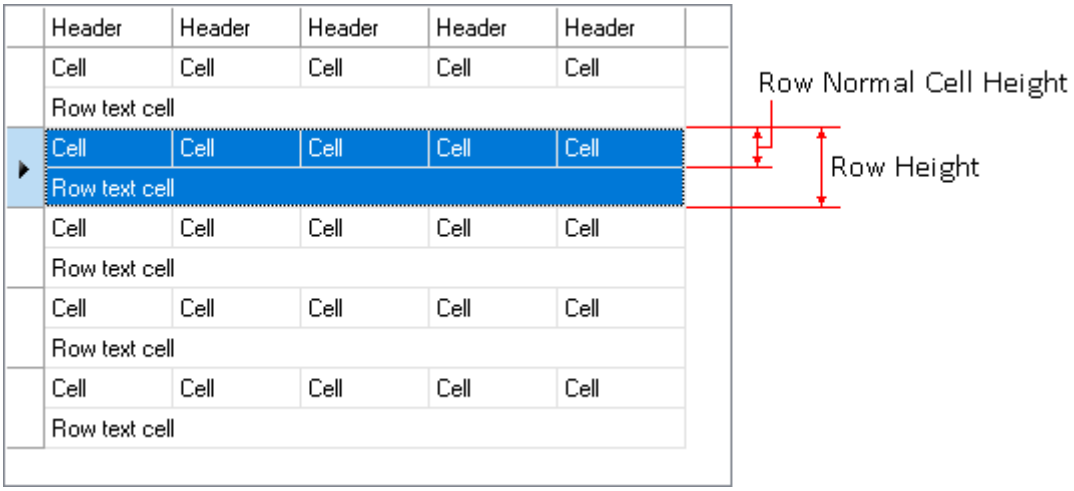
You access the row text column with the **RowTextCol** property of iGrid. An equivalent way to access this column in code is to use the index of -1 in the **Cols** collection. The data of some events (such as **CellClick**) return the column index of -1 if the event has been triggered for a cell from the row text column (for instance, a group row). You can pass the index of -1 to some methods as the argument representing an iGrid column. For example, you can group iGrid by the values of the row text column using the following code:

```
iGrid1.GroupObject.Add(-1);
iGrid1.Group();
```

Despite all these special features of the row text column, the cells of the row text column are normal cells with all their traditional cell properties (i.e. they are objects of the **iGCell** type). You can specify the contents of group rows and row text cells using the **Value** and **ImageIndex** properties, set the background and foreground colors using the **BackColor** and **ForeColor** properties, and so on.

16.2. Row Text Cells

Every iGrid row can have a special wide cell spanning all or several adjacent columns and placed under normal cells. This cell is called "row text cell". It can be used to display additional row data similar to message preview in the message list in Microsoft Outlook. Below is a screenshot illustrating how row text cells look:



An iGrid row is represented with an **iGRow** object. Its **Height** property determines the height of the whole row, and the **NormalCellHeight** property stores the height of normal cells. The remaining height is used to display the row text cell.

Row text cells are disabled by default. To enable them, do the following:

1. Set the **RowMode** property to True because row text cells can be displayed only in row mode.
2. Set the **RowTextVisible** property to True to make row text cells visible. The default value of this property is False and row text cells are not displayed.
3. Set the **Height** and **NormalCellHeight** properties accordingly to provide space for row text cells. The height of the row text cell in a row is the difference between the values of these properties and may vary from row to row.

To specify the properties of row text cells, use the **Cells** property of the row text column (**RowTextCol**) or the **RowTextCell** property of the **iGRow** object.

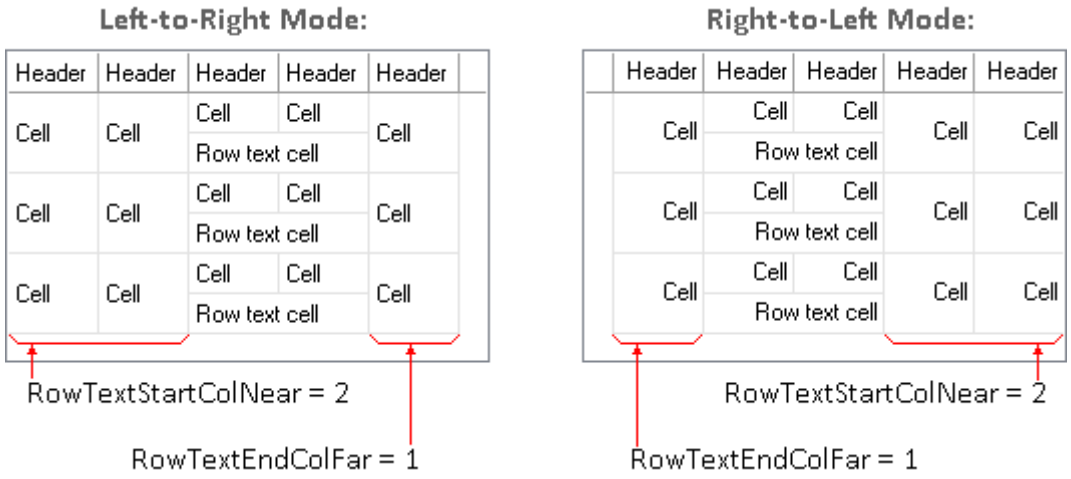
Below is a simplest example of displaying a row text cell in one row. The code snippet below enables row texts in iGrid1, makes the row text cell visible in the first row, and sets a bold row text:

```
iGrid1.RowMode = true;
iGrid1.RowTextVisible = true;

iGrid1.Rows[0].NormalCellHeight = 20;
iGrid1.Rows[0].Height = 40;

iGrid1.Rows[0].RowTextCell.Value = "Row text cell";
iGrid1.Rows[0].RowTextCell.Font = new Font(iGrid1.Font, FontStyle.Bold);
```

As a rule, all rows have the same layout. If you need rows with row text cells of the same structure, you can specify the corresponding default settings before adding rows and make your code short and effective. Below is an example of such a code snippet based on the **DefaultCol** and **DefaultRow** properties of iGrid. This is the code that was used to create the grid on the screenshot above:

```
iGrid1.RowTextVisible = true;
iGrid1.RowMode = true;
iGrid1.RowHeader.Visible = true;

iGrid1.DefaultCol.Text = "Header";
iGrid1.DefaultCol.DefaultCellValue = "Cell";
iGrid1.DefaultRow.NormalCellHeight = 20;
iGrid1.DefaultRow.Height = 40;
iGrid1.RowTextCol.DefaultCellValue = "Row text cell";

iGrid1.Cols.Count = 5;
iGrid1.Rows.Count = 5;
```

By default iGrid draws row text cells starting from the first visible column till the last visible column. The **RowTextStartColNear** property allows you to specify the column each row text cell starts in. In fact, its value specifies the number of columns skipped from the left (or from the right in right-to-left mode). The similar **RowTextEndColFar** property allows you to specify the column from the right row text cells end in. Note that counting starts from the right for this property (or from the left in right-to-left mode). The following figures explain how these properties work:



Two final notes regarding grids with row text cells:

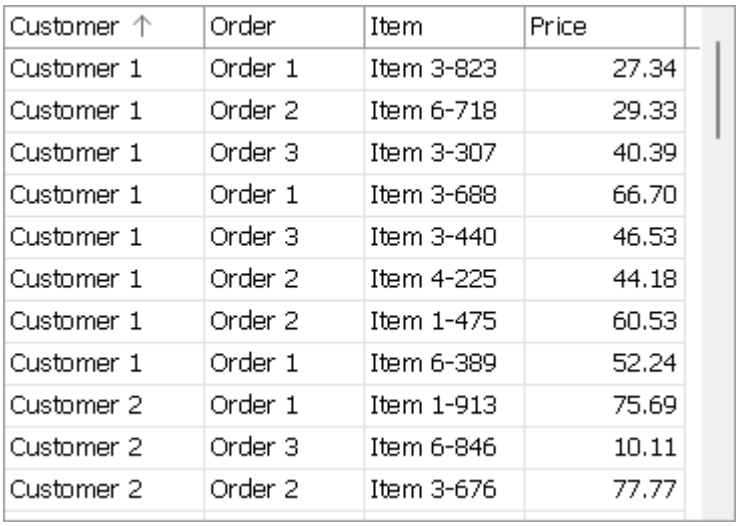
- The user can freely reorder columns, and the row text cells will visually remain on their places.
- You cannot freeze some first columns so that the frozen area edge would cross row text cells. If you need row text cells in a grid with frozen columns, you can specify the first column for the row text cells in the non-frozen area with the **RowTextStartColNear** property iGrid.

17. SORTING

17.1. Interactive Sorting

iGrid allows you to sort its rows by one or several columns using various criteria.

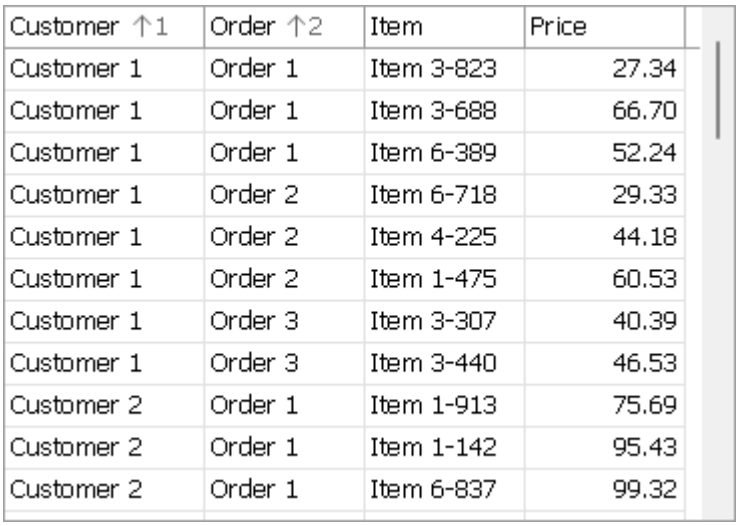
To sort iGrid by a column, click its header. iGrid will add the arrow glyph indicating the sort order of the column to its header:



Customer ↑	Order	Item	Price
Customer 1	Order 1	Item 3-823	27.34
Customer 1	Order 2	Item 6-718	29.33
Customer 1	Order 3	Item 3-307	40.39
Customer 1	Order 1	Item 3-688	66.70
Customer 1	Order 3	Item 3-440	46.53
Customer 1	Order 2	Item 4-225	44.18
Customer 1	Order 2	Item 1-475	60.53
Customer 1	Order 1	Item 6-389	52.24
Customer 2	Order 1	Item 1-913	75.69
Customer 2	Order 3	Item 6-846	10.11
Customer 2	Order 2	Item 3-676	77.77

To toggle the sort order of the sorted column, click its header again. To sort iGrid by another column, click the header of the new sort column.

To sort the grid by several columns, click the corresponding column headers holding down the CTRL key. iGrid will add numbers indicating the order of the column in the sort criteria after the sort arrows:



Customer ↑1	Order ↑2	Item	Price
Customer 1	Order 1	Item 3-823	27.34
Customer 1	Order 1	Item 3-688	66.70
Customer 1	Order 1	Item 6-389	52.24
Customer 1	Order 2	Item 6-718	29.33
Customer 1	Order 2	Item 4-225	44.18
Customer 1	Order 2	Item 1-475	60.53
Customer 1	Order 3	Item 3-307	40.39
Customer 1	Order 3	Item 3-440	46.53
Customer 2	Order 1	Item 1-913	75.69
Customer 2	Order 1	Item 1-142	95.43
Customer 2	Order 1	Item 6-837	99.32

To toggle the sort order of the sorted column in multi-column sort, click its header again holding down the CTRL key.

By default, columns are sorted by cell values. The **SortType** property of a column (the **iGCol** object) allows you to specify other available sort criteria, such as sort by cell image or foreground color. This property accepts values from the **iGSortType** enumeration reflecting all available sort rules. One of its items, **Custom**, can be used to enable custom sorting in a column. Examples of custom sorting can be found in the topic dedicated to the **CustomSort** event and in the Custom Sorting and Sorting of Non-string Data Stored as Strings topic in this chapter.

A column is sorted in the ascending order by default. You can change the default sort order to descending with the **SortOrder** property of the **iGCol** class.

17.2. Sorting iGrid from Code

To sort iGrid from code, do the following:

1. Specify the sort criteria in the **SortObject** property of iGrid.
2. Invoke the **Sort** method of iGrid to perform actual sorting.

The instance of the **iGSortObject** class returned by the **SortObject** property stores the current sorting information. It is a collection of **iGSortItem** objects. Every **iGSortItem** object represents an iGrid column used in the sort criteria, and its main **SortOrder** and **SortType** properties are used to specify the sort order and sort type for the column.

To add new columns to the **SortObject**, usually one of the overloaded versions of its **Add** method is used. For example, to sort iGrid by the third column, do the following:

```
iGrid1.SortObject.Clear();
iGrid1.SortObject.Add(2);
iGrid1.Sort();
```

This will sort the grid by the cell values (the default sort type) of the third column in the ascending order (the default sort order).

If you would need to sort the grid by the foreground colors of the cells in the second column from the very light color to the darkest one, your code could look like this:

```
iGrid1.SortObject.Clear();
iGrid1.SortObject.Add(1);
iGrid1.SortObject[0].SortType = iGSortType.ByForeColor;
iGrid1.SortObject[0].SortOrder = iGSortOrder.Descending;
iGrid1.Sort();
```

Or, in a shorter form:

```
iGrid1.SortObject.Clear();
iGrid1.SortObject.Add(1, iGSortOrder.Descending, iGSortType.ByForeColor);
iGrid1.Sort();
```

iGrid's **SortObject** is used in two directions:

1. If you need to sort the grid from code, you define the sort criteria in the **SortObject** and issue the **Sort** method.
2. If the user has sorted the grid by clicking its column headers, the old contents of the **SortObject** are cleared and its new state reflects the current sort criteria.

Note that changing the contents of the **SortObject** in code does not sort iGrid — you only define a sort criteria. Actual sorting is performed when you call the **Sort** method.

17.3. Common Sorting Tasks

Displaying column sort state

iGrid with the default settings displays sort arrows and numbers in the headers of the sorted columns to help you to determine the sort order and sort index of each column. This information is also called "sort info". You can disable displaying of sort info in a column header with the **SortInfoVisible** property of the corresponding **iGColHdr** object, for example:

```
fGrid.Header.Cells[0, 0].SortInfoVisible = iGBool.False;
```

This can be useful if you create narrow columns that do not have enough space to show column header title and sort info together. The fact is that iGrid draws the sort info in a column header first, and only after that draws the text and image in the remaining area. If a column is narrow enough, the sort info can occupy the whole available space and there will be no place for the column title.

Main events related to sorting

iGrid raises the **BeforeContentsSorted** event before sorting and **AfterContentsSorted** after sorting. They are particularly useful in scenarios where auxiliary rows, such as subtotal rows at the bottom of each group, are added to the grid. For example, you might want to temporarily remove these rows prior to sorting to avoid interference with the sort operation, and then reinsert them once sorting is complete.

Protecting columns from being sorted by the user

There are three techniques you can use to disable interactive sorting of columns:

- Set the **AllowPress** property of the **Header** object to False. All column headers will become non-clickable and will not even indicate the pressed state.
- Set the sort type of a column (the **SortType** property) to **None**.
- Handle the **ColHdrClick** event. The **DoDefault** field of the event's data can be set to False to disable sorting of a column dynamically based on a condition.

Saving and restoring sorting state

This task can be easily implemented using the **LayoutObject** property of iGrid. For a detailed description and example, see Saving and Restoring Grid Layout.

17.4. Stay-sorted Mode

You can activate a special stay-sorted mode that keeps iGrid sorted even after interactive changes to cell values. When this mode is enabled, any modification to a cell that affects the current sort criteria will automatically reposition its row based on the updated sort order. To enable this behavior, set the **StaySorted** property of iGrid to True.

Note that:

- The stay-sorted mode does not work with columns sorted by selection.
- Rows cannot be moved in the stay-sorted mode.
- You cannot turn the stay-sorted mode on if the grid already contains group rows.
- You cannot add group rows to iGrid from code in the stay-sorted mode.

17.5. Unsortable Rows

In iGrid, a row is excluded from sorting if:

- The row is a group row (its type is **ManualGroupRow** or **AutoGroupRow**).
- The **Sortable** property of the row is set to False.
- The row belongs in the frozen area and the **SortFrozenRows** property of the **FrozenArea** object property is set to False.

Such unsortable rows do not change their position in the grid during sorting. Contiguous sets of rows between unsortable rows are sorted independently as if each set of rows were separate grids. This behavior allows you to sort the grid properly when the grid is grouped and normal rows inside each group should be sorted within the bounds of the group.

When you sort the grid with the overloaded version of the **Sort** method without parameters, all grid rows except unsortable ones (if any) are included in the sorting. There are two overloaded versions of the **Sort** method that can be used to specify only a contiguous subset of rows to sort. This feature can be useful, for instance, when you insert a contiguous bunch of rows into the sorted grid and you want to keep the grid in sorted mode. In this case you do not need to resort the entire grid, you should do it only for the set of the newly added rows. Those overloaded versions of the **Sort** method will help you with it.

17.6. Custom Sorting and Sorting of Non-string Data Stored as Strings

iGrid supports developer-defined sorting rules for cell values. This is especially useful when non-string data — like numbers or dates — are retrieved from a source that stores them as strings. Without custom sorting, such values are treated and ordered as strings, which can result in incorrect sort behavior.

Custom sorting can help to solve this problem. Let's create a test grid in which integer values from 1 to 100 are stored as strings:

```
iGrid1.Cols.Count = 1;
iGrid1.Cols[0].Text = "Numbers";
iGrid1.Rows.Count = 500;

Random rnd = new Random(123);
foreach (iGCell cell in iGrid1.Cols[0].Cells)
{
  cell.Value = rnd.Next(1, 1001).ToString();
}
```

If we click the column header of the Numbers column to sort the grid by the cell values in this column, the rows will be sorted not in the order expected by the human:

To fix this problem, we will use custom sorting. We should specify that the column will use custom sorting by setting the **SortType** property of the column to **Custom** and attach an event handler performing correct comparison to the **CustomSort** event of iGrid:

```
iGrid1.Cols[0].SortType = iGSortType.Custom;
iGrid1.CustomSort += iGrid1_CustomSort;
```

The event handler itself could look like this:

```
private void iGrid1_CustomSort(object sender, iGCustomSortEventArgs e)
{
  if (e.ColIndex == 0)
  {
    int val1 = Convert.ToInt32(e.Value1);
    int val2 = Convert.ToInt32(e.Value2);
    e.Result = val1.CompareTo(val2);
  }
}
```

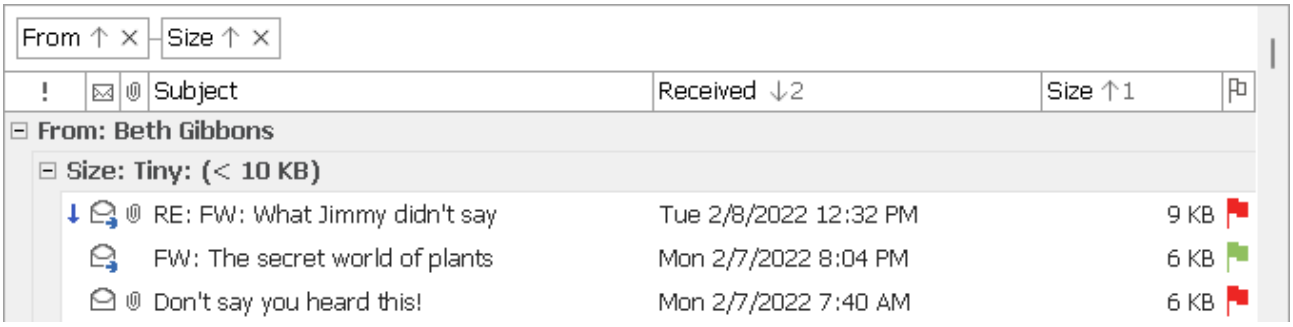
Now the grid is sorted as expected when we click the column header:

18. GROUPING

18.1. Group Box and Interactive Grouping

The group box is an optional area above column headers intended to group rows interactively:

You can group rows by dragging column headers into this area.



To remove a column from grouping, drag its column header from the group box back to the line of column headers. An alternative way is to click the close button with the 'X' glyph. These X-buttons are added to column headers automatically when column headers are in the group box. Removal of a column from grouping using dragging allows you to place the column into any place. Clicking the X-button automatically places the column onto its original place.

When a column header is in the group box, you can click it to sort the grid by the corresponding column. In this case group rows are sorted accordingly.

The group box can be adjusted with the help of the **GroupBox** object property of iGrid. The following table lists the properties of the **GroupBox** object you can use to adjust the group box:

PROPERTY	DESCRIPTION
BackColor	Gets or sets the color of the group box's background.
ColHdrBorderColor	Gets or sets the color of the column header borders and the lines connecting them in the group box.
ColHdrSpace	Gets or sets the size of empty space around a column header in the group box.
ColHdrXButtons	Gets or sets a value indicating whether X-buttons are displayed inside column headers placed in the group box.
ConnectorLines	Gets or sets a value indicating whether the lines connecting column headers in the group box are drawn.

PROPERTY	DESCRIPTION
HintBackColor	Gets or sets the background color of the text shown in the group box when it is empty.
HintForeColor	Gets or sets the color of the text shown in the group box when it is empty.
Layout	Gets or sets the layout mode for column headers inside the group box (steps or row).
Visible	Determines whether the group box is visible.

The group box isn't visible by default. To display it, set the **Visible** property of the **GroupBox** object to True:

```
iGrid1.GroupBox.Visible = true;
```

If you need to know the height of the group box, you can retrieve it with the read-only **Height** property of the **GroupBox** object.

18.2. Grouping and Ungrouping Rows from Code

Grouping from code is very similar to sorting. You define the grouping criteria in the **GroupObject** property of iGrid and then call iGrid's **Group** method to perform actual grouping. The **GroupObject** property returns an instance of the **iGSortObject** class like the **SortObject** property used to define sorting criteria, and you define the set of columns to group by in the same manner as you do it with the **SortObject** when you sort the grid.

Changes in the **GroupObject** do not affect the actual grouping until you call the **Group** method. The group box on the screen (if visible) is updated accordingly after you call this method. The opposite is also true: when the user groups the grid using the group box, the **GroupObject** is updated to reflect the current grouping.

After you call the **Group** method, iGrid creates so-called automatic group rows accordingly to the defined grouping criteria. The **Type** property of these rows is set to **AutoGroupRow**. The height of these rows can be specified with the **DefaultAutoGroupRow** property of iGrid. **DefaultAutoGroupRow** also allows you to determine the initial state of the group rows created automatically — collapsed or expanded.

The following example shows how to group a grid by a column with the index 1:

```
//Add the column to the group object
iGrid1.GroupObject.Add(1);

//Perform actual grouping
iGrid1.Group();
```

The next example shows you how to group a grid by two columns with the keys From and Flag:

```
//Add the columns to the group object
iGrid1.GroupObject.Add("From");
iGrid1.GroupObject.Add("Flag");

//Perform actual grouping
iGrid1.Group();
```

To ungroup a grid from code (i.e. to remove grouping), clear the **GroupObject** and call the **Group** method of iGrid:

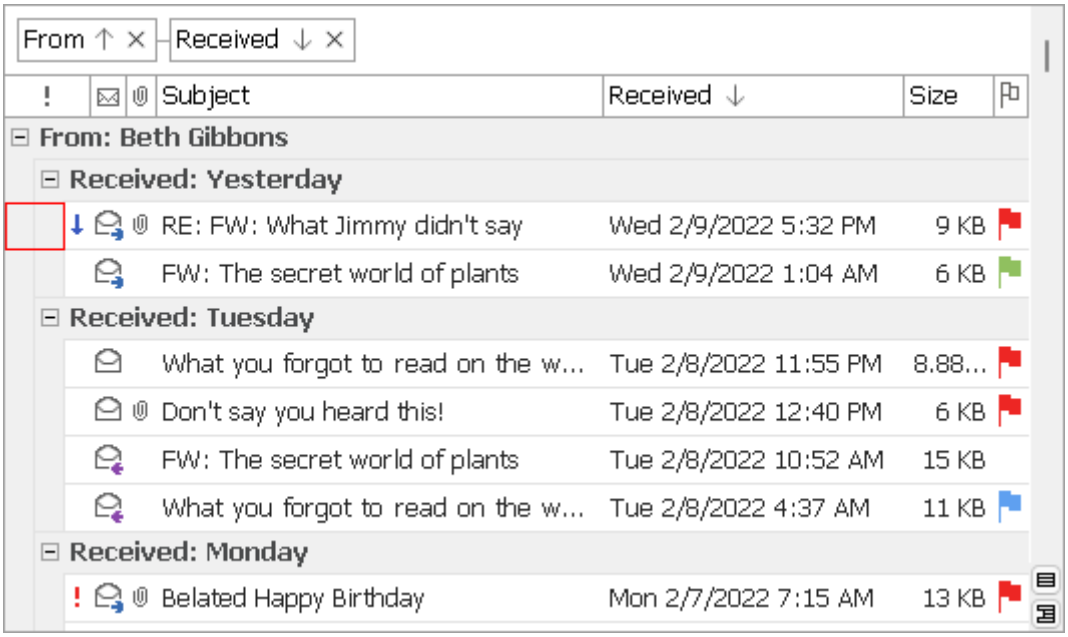
```
iGrid1.GroupObject.Clear();
iGrid1.Group();
```

18.3. Row Levels and Level Indents

Grouping in iGrid is based on the concept of row levels. Every row has a level that determines hierarchical relations between rows. Top-level rows have level 0, their child rows have level 1, the child rows for the first-level child rows have level 2, and so on.

Before grouping iGrid does not have group rows and the row levels for all rows with normal cells are equal to 0. After grouping iGrid creates automatic group rows with levels 0, 1, 2 and so on according to the grouping criteria. The levels of all rows with normal cells are set to the value representing the deepest level, i.e. the number of columns you grouped iGrid by.

To represent hierarchical relations between rows with different levels for the user, iGrid draws its rows with indents. In the terminology of iGrid these indents are called "row level indents" or simply "level indents". Every row has the special level indent area at the left consisting of typical level indents repeated for every hierarchy level. The red rectangle on the following picture marks the level indent area for the third row:



The very first row on the screenshot has level 0, the second row has level 1 (it is a child row of the previous top-level group row), and the third row has level 2. The first row is drawn without indent because its level is 0, the second row is drawn with 1 level indent, the third row is drawn with 2 level indents, which correspond to the values of levels for these rows.

The value of level for a particular row is stored in the **Level** property of the corresponding **iGRow** object. As a rule, you do not change these values set by iGrid automatically, but you can do this if you construct a specific iGrid row by row manually from code.

The size (width) of level indent is determined by iGrid automatically and can be retrieved with the **LevelIndent** property of iGrid. In some special cases you may need to change this value. This can be done with the **LevelIndentOverride** property of iGrid. For more information, read the Concept of Overridable Properties topic in this manual.

Drawing of level indents can be redefined with the **CustomDrawLevelIndentPart** event. The Custom-Drawn Level Indents topic describes this concept with an accompanying example.

18.4. Formatting of Automatic Group Rows by Levels

When an automatic group row is created, its formatting is set based on the **GroupRowLevelStyles** property of iGrid. This property stores an array of style objects (**iGCellStyle**) used to format group rows by levels. The styles are assigned to the **CellStyle** property of the **iGRow** object that represents an automatic group row after its creation.

Each element of the array is used as the style of the corresponding level of grouping. For example, when the grid is being grouped by two columns, the first element of the **GroupRowLevelStyles** array is used as the style of the group rows created for the first column, and the second element — as the style of the group rows created from the second column. If the length of the array is less than the number of grouped columns, then the style object of a group row from nested levels that do not have the corresponding element in the **GroupRowLevelStyles** array will refer the style from the last element of the array. For example, if the grid is being grouped by three columns, but **GroupRowLevelStyles** contains two elements, its first element will be used as the style of top-level group rows, and its second element will be used as the style for the group rows created for the second and third columns.

By default the **GroupRowLevelStyles** array contains one element, and the styles of all group rows from all levels refer to this style object. From this point of view the only element of **GroupRowLevelStyles** can be considered as the default formatting object for all automatic group rows.

If you want to use different formatting for different levels, create the corresponding number of elements in the array and set the required properties. Below is a C# example of how to set the background color of all group rows on the first level to a blue color and the background color of all group rows on the second and deeper levels to a brown color:

```
iGrid1.GroupRowLevelStyles = new iGCellStyle[2];
iGrid1.GroupRowLevelStyles[0] = new iGCellStyle();
iGrid1.GroupRowLevelStyles[0].BackColor = Color.AliceBlue;
iGrid1.GroupRowLevelStyles[1] = new iGCellStyle();
iGrid1.GroupRowLevelStyles[1].BackColor = Color.SandyBrown;
```

The equivalent VB code will slightly differ because of another way of working with arrays:

```
Dim myCellStyleArray(1) As iGCellStyle
myCellStyleArray(0) = New iGCellStyle()
myCellStyleArray(0).BackColor = Color.AliceBlue
myCellStyleArray(1) = New iGCellStyle()
myCellStyleArray(1).BackColor = Color.SandyBrown
iGrid1.GroupRowLevelStyles = myCellStyleArray
```

Pay attention to the fact that the style objects of group rows refer to the objects in the **GroupRowLevelStyles** array. This allows you to change the look of all existing group rows simply by changing properties of the corresponding style object in the array. For example, if you have only one element in the **GroupRowLevelStyles** array (the default case), you can make the font of all group row texts bold after you created them with the following code:

```
iGrid1.GroupRowLevelStyles[0].Font = new Font(iGrid1.Font, FontStyle.Bold);
iGrid1.Invalidate();
```

iGrid implements the **GetDefaultGroupRowLevelStyles** method that returns the default contents of its **GroupRowLevelStyles** array. You can use the value returned by this method to reset the **GetDefaultGroupRowLevelStyles** property to its default value.

As written above, **GetDefaultGroupRowLevelStyles** has one element. It is an instance of the **iGCellStyle** class in which the **BackColor** property is set to **SystemColors**.**Control**. All other

properties are set to special "NotSet" values allowing property value inheritance from column cell styles.

In most cases the cell style objects from the **GroupRowLevelStyles** array are used "as is" to format the corresponding levels of group rows. However, for some sort types iGrid clones the cell styles and changes their properties to correspond to the sort type. For example, if the sort type of a column is **ByForeColor**, the value of the **ForeColor** property of the resulting group style object will be set to the foreground color for which the corresponding group has been created. You can find out more about this in the Contents of Automatic Group Rows topic.

18.5. Contents of Automatic Group Rows

When the user groups data in iGrid by dragging column headers to the group box or the developer does this from code, iGrid creates automatic group rows with corresponding contents and sets their visual properties, such as background color. The process of visual formatting of automatic group rows is described in the Formatting of Automatic Group Rows by Levels topic. This topic is focused mainly on the process of automatic creation of group row contents, namely group titles and accompanying images.

A group title is some text that describes a set of rows the group row was created for. A group title is built from the following parts:

1. Column title prefix (optional).
2. Group value text, which is the text representation of the value for which the group was created.
3. Item count, showing the number of items in the group.
4. One or more summary values (if defined).

The **GroupTitleOptions** object property of iGrid allows you to control the first and third parts. It implements 2 sub-properties — **AddColTitlePrefix** and **ItemCountFormatString**. If the Boolean **AddColTitlePrefix** property is set to True (the default value), the column title of the grouped column will be added before the group value text as "<Title>:". The **ItemCountFormatString** property is a string property allowing you to specify the format of the item count text added after the group value text. It is set to "({0})" by default, which means that the number of items in parenthesis will be added after the group value text. You can set this property, for example, to "({0} items)" to add the "items" word. To suppress the item count in group titles, set this property to null.

The fourth part, summary values, is controlled with the help of the **GroupTitleSummaries** object property of iGrid. It is described in greater detail in the Group Title Summaries topic in this manual.

The second part, group value text, depends on the column sort type (the **SortType** property of the **iGCol** class). By default columns are sorted by cell values, and group value texts are cell texts. For other sort types group value texts are determined differently. The table below explains how iGrid determines group value text together with group image and group row style for all possible sort types:

COLUMN SORT TYPE	GROUP VALUE	GROUP VALUE TEXT	GROUP IMAGE INDEX	GROUP STYLE
ByValue	<Cell Value>	<Cell Text>	-1	A reference to a style from the **GroupRowLevelStyles** property

COLUMN SORT TYPE	GROUP VALUE	GROUP VALUE TEXT	GROUP IMAGE INDEX	GROUP STYLE
ByAuxValue	<Cell Auxiliary Value>	<Cell Auxiliary Value>.**ToString**()	-1	A reference to a style from the **GroupRowLevelStyles** property
ByText	<Cell Text>	<Cell Text>	-1	A reference to a style from the **GroupRowLevelStyles** property
ByTextNoCase	<Cell Text>.**ToUpper**()	<Cell Text>.**ToUpper**()	-1	A reference to a style from the **GroupRowLevelStyles** property
ByImageIndex	null	null	<Cell Image Index>	A reference to a style from the **GroupRowLevelStyles** property
ByForeColor	<Cell Foreground Color>	<Cell Foreground Color>.**ToString**()	-1	A clone of the style from **GroupRowLevelStyles** with the **ForeColor** property set to <Cell Foreground Color>
ByBackColor	<Cell Background Color>	null	-1	A clone of the style from **GroupRowLevelStyles** with the **BackColor** property set to <Cell Background Color>
ByFont	<Cell Font>	Short font description (name, size, style)	-1	A reference to a style from the **GroupRowLevelStyles** property
BySelected	True | False	"Selected" | "Not Selected"	-1	A reference to a style from the **GroupRowLevelStyles** property
Custom	<Cell Text>	<Cell Text>	-1	A reference to a style from the **GroupRowLevelStyles** property

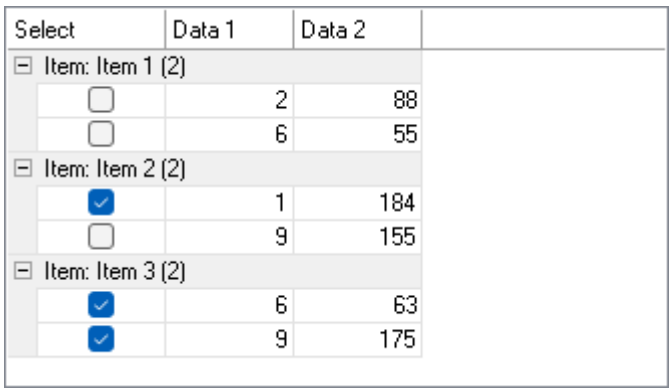
Notes to the table:

- The group value text is set to null for the **ByImageIndex** and **ByBackColor** sort types (in other words, it is always absent). Because of this, the column title is never added as prefix in for these sort types even if **GroupTitleOptions**.**AddColTitlePrefix** is set to True.
- The "Selected" and "Not Selected" strings for the **BySelected** sort type can be changed/localized with the **GroupValueTextSelected** and **GroupValueTextNotSelected** properties of the **UIStrings** object property of iGrid.

The raw group value, group value text, item count and calculated summary values for the group are stored in the **GroupInfo** object property of the group row (the **iGRow** object). The group value text and image index are saved in the **Value** and **ImageIndex** properties of the row text cell of the group row (this concept is described in the Row Text Column topic). You can access all these values in event handlers of the **AfterAutoGroupRowCreated** and **CustomGroupTitle** events of iGrid to retrieve information about group rows and to modify them if needed. The capabilities of these events, along with examples of their use, can be found in the Custom Group Titles and Individual Adjustment of Automatic Group Rows topics.

18.6. Custom Group Titles

If the built-in group title configuration options described in the Contents of Automatic Group Rows topic are not sufficient, you can always implement your own group title building algorithm using the **CustomGroupTitle** event. As an example of its use, let's consider the following grid:



It's a grid with one check box and two numeric data columns grouped by the Item column. The source code of this grid might look like this:

```
iGrid1.HighlightSelCells = false;

iGrid1.Cols.Add("item", "Item");
iGrid1.Cols.Add("select", "Select");
iGrid1.Cols.Add("data1", "Data 1");
iGrid1.Cols.Add("data2", "Data 2");

iGCellStyle cs = iGrid1.Cols["select"].CellStyle;
cs.Type = iGCellType.Check;
cs.ImageAlign = iGContentAlignment.MiddleCenter;

iGrid1.Cols["data1"].CellStyle.TextAlign = iGContentAlignment.MiddleRight;
iGrid1.Cols["data2"].CellStyle.TextAlign = iGContentAlignment.MiddleRight;

iGrid1.Rows.Count = 6;

string[] items = { "Item 1", "Item 2", "Item 3" };
Random random = new Random(124);

foreach (iGRow row in iGrid1.Rows)
{
  row.Cells["item"].Value = items[row.Index % items.Length];
  row.Cells["select"].Value = random.Next(2) == 0;
  row.Cells["data1"].Value = random.Next(10);
  row.Cells["data2"].Value = random.Next(200);
}

iGrid1.GroupObject.Add("item");
iGrid1.Group();
```

Suppose you want to display how many rows are selected out of the total number of rows in a group in the group headers (selection is made using the check boxes in the rows). To solve this task, let's do the following two things before we call the **Group** method of iGrid that performs data grouping.

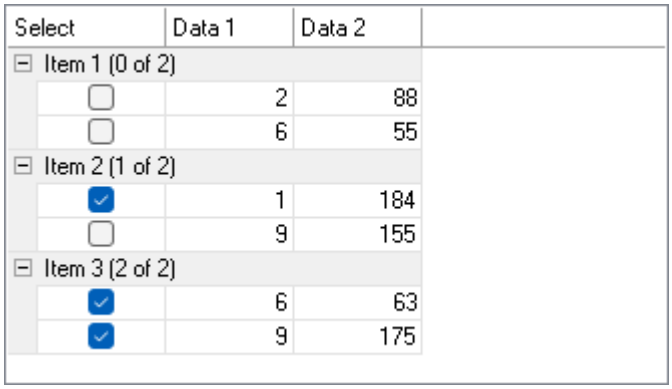
First, we define the invisible group title summary that calculates the number of selected rows. The **Sum** function can be used with check box cells for that:

```
iGrid1.GroupTitleSummaries.Add("sel", "select", iGSummaryType.Sum);
```

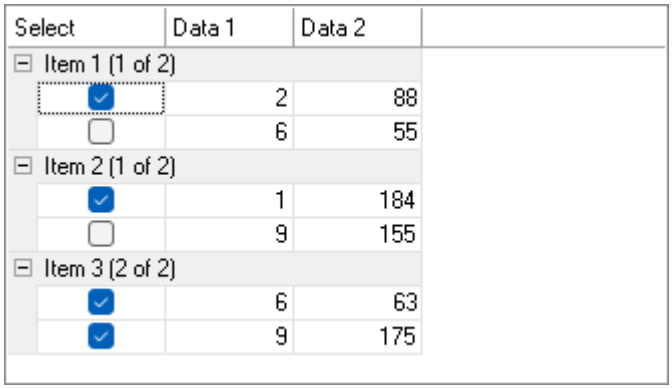
After that we connect the following event handler to the **CustomGroupTitle** event of our iGrid:

```
private void iGrid1_CustomGroupTitle(
  object sender, iGCustomGroupTitleEventArgs e)
{
  var groupValueText = e.GroupInfo.GroupValueText;
  var selectedCount = e.GroupInfo.SummaryValues["sel"];
  var totalCount = e.GroupInfo.ItemCount;
  e.GroupTitle = $"{groupValueText} ({selectedCount} of {totalCount})";
}
```

Now we will see the following grid after launching our modified code:



It is important to note that iGrid automatically updates summary values when its cells are changed. This also triggers the **CustomGroupTitle** event, in which group titles will be updated accordingly. You can verify this by checking the check box in the first row:



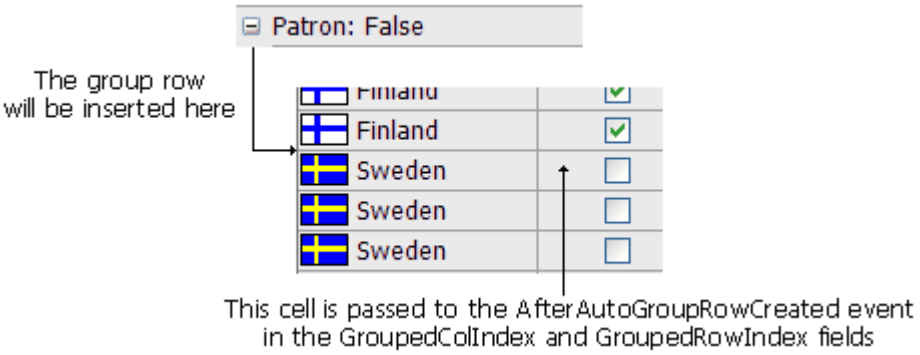
The **AfterAutoGroupRowCreated** event can also be used to modify group titles and set other custom formatting parameters for group rows. However, please note that **AfterAutoGroupRowCreated** is not raised when grid cells are changed, as is the case with **CustomGroupTitle**. For more information about the capabilities of the **AfterAutoGroupRowCreated** event, read the Individual Adjustment of Automatic Group Rows topic.

18.7. Individual Adjustment of Automatic Group Rows

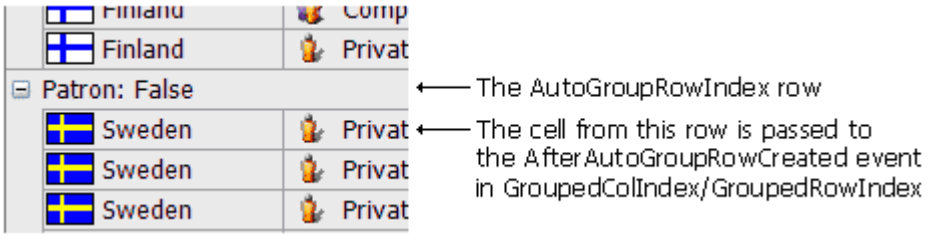
You can set individual formatting or add custom content (such as image) in any automatic group row after it has been created with the **AfterAutoGroupRowCreated** event. It provides the following event arguments:

- **AutoGroupRowIndex** — the index of the created automatic group row;
- **GroupedColIndex** and **GroupedRowIndex** — the column and row indexes respectively of the cell from the first child row of the group in the column the grid is being grouped by.

To explain the values passed by iGrid in these arguments, let's consider a grid with a Patron column containing Boolean values. If you group the grid by this column, the grid is sorted by this column first and then group rows are inserted. The picture below demonstrates the place where the "Patron: False" group row will be inserted and which cell will be passed to the **AfterAutoGroupRowCreated** event in the **GroupedColIndex** and **GroupedRowIndex** fields:



The Patron column becomes hidden after grouping, but you can still access its cells through the **Cells** collection of iGrid in the **AfterAutoGroupRowCreated** event:



You can use these values to adjust the group row. This is done with the corresponding row text cell containing the group title and image, and the row's cell style object. For example, you can place the image from the first group cell into the group row itself with an event handler like this:

```
private void fGrid_AfterAutoGroupRowCreated(
  object sender, iGAfterAutoGroupRowCreatedEventArgs e)
{
  if (fGrid.Cols.GetKey(e.GroupedColIndex) == "Country")
  {
    // Show the image from the first cell in the group.

    iGCell myGroupTitleCell =
      fGrid.RowTextCol.Cells[e.AutoGroupRowIndex];
    iGCell myFirstCellInGroup =
      fGrid.Cells[e.GroupedRowIndex, e.GroupedColIndex];

    myGroupTitleCell.ImageList =
      fGrid.Cols[e.GroupedColIndex].CellStyle.ImageList;
    myGroupTitleCell.ImageIndex =
      myFirstCellInGroup.ImageIndex;
  }
}
```

If you want to change formatting parameters of a whole automatic group row (such as background color or font), you can't do this simply by changing the corresponding properties of the row's **CellStyle** object property. The fact is that this property contains a reference to the style object from the **GroupRowLevelStyles** property of iGrid, and this object is used in all automatic group rows of the same hierarchy level (see the Formatting of Automatic Group Rows by Levels topic for more information). If you change the properties of this cell style, you will change formatting of all group rows belonging to this hierarchy level. To do it right, you need to clone the style object first, and change its formatting properties only after that. Let's demonstrate a wrong and right approaches of doing this.

Imagine a grid in which the first column contains integer values. When the user groups the grid by the values of this column, you want to highlight group rows created for values exceeding 50 with a

variation of a red color. An incorrect way to do this using the **AfterAutoGroupRowCreated** event might look like this:

```
private void iGrid1_AfterAutoGroupRowCreated(
  object sender, iGAfterAutoGroupRowCreatedEventArgs e)
{
  if (e.GroupedColIndex == 0)
  {
    if ((int)e.GroupInfo.GroupValue > 50)
    {
      iGrid1.Rows[e.AutoGroupRowIndex].CellStyle.BackColor =
        Color.PaleVioletRed;
    }
  }
}
```

Every created automatic group row is formatted using the same style object stored in the first element of the **GroupRowLevelStyles** array, and changing its **BackColor** property actually results in changing the background color of all group rows. To implement our task properly, we need to clone the group row style with the **Clone** method of the **iGCellStyle** object before setting the **BackColor** property of the cell:

```
private void iGrid1_AfterAutoGroupRowCreated(
  object sender, iGAfterAutoGroupRowCreatedEventArgs e)
{
  if (e.GroupedColIndex == 0)
  {
    if ((int)e.GroupInfo.GroupValue > 50)
    {
      iGCellStyle myCellStyle =
        iGrid1.Rows[e.AutoGroupRowIndex].CellStyle.Clone();
      myCellStyle.BackColor = Color.PaleVioletRed;
      iGrid1.Rows[e.AutoGroupRowIndex].CellStyle = myCellStyle;
    }
  }
}
```

There is one potential drawback in the code above. If our grid has many groups with values greater than 50, we will create many clones of the same style object with the same property values and consume memory irrationally. A better approach would be creation of a style object with our special formatting and use it in the **AfterAutoGroupRowCreated** event handler. For example, we could define such a formatting object in form's module as follows:

```
private iGCellStyle fGroupRowStyle;
```

Then initialize it in the form's **Load** event handler like this:

```
fGroupRowStyle = iGrid1.GroupRowLevelStyles[0].Clone();
fGroupRowStyle.BackColor = Color.PaleVioletRed;
```

And finally, use it in the **AfterAutoGroupRowCreated** event handler:

```
private void iGrid1_AfterAutoGroupRowCreated(
  object sender, iGAfterAutoGroupRowCreatedEventArgs e)
{
  if (e.GroupedColIndex == 0)
  {
    if ((int)e.GroupInfo.GroupValue > 50)
    {
        iGrid1.Rows[e.AutoGroupRowIndex].CellStyle = fGroupRowStyle;
    }
  }
}
```

Do not use the **AfterAutoGroupRowCreated** event to modify group titles. A group title can change many times after an automatic group row has been created. This happens, for example, when group summaries are recalculated. The **AfterAutoGroupRowCreated** event is raised only once, immediately after the group row is created. Use the **CustomGroupTitle** event to customize group titles instead. For more information, see the Custom Group Titles topic.

18.8. Grouping by Custom Values (Range Grouping)

When iGrid is grouping rows by a column, it retrieves group values for each cell in the column. These group values are used to create groups. The default sort type for a column is **iGSortType**.**ByValue**, and the group value of a cell equals to the value of its **Value** property. In iGrid, you can specify your own group values for cells and get any desirable set of groups defined by these values.

One of the cases when this feature is useful is grouping cell values by ranges. A good example of this is Microsoft Outlook that allows you to group messages by size categories — such as Tiny (< 10 KB), Small (10-25 KB), Medium (25-100 KB), Large (100-500 KB) and so on. These group values are determined for each message dynamically, while Outlook still operates with message sizes in bytes.

To create a column with custom grouping in iGrid, do the following:

1. Set the **CustomGrouping** property of the corresponding **iGCol** column object to True.
2. Handle the **CustomGroupValue** event of iGrid and provide the custom group value for each cell in the column in the **Value** field of the event arguments.

As an example, let's consider a grid with a Sales column containing sale amounts by customers. When the user will group the grid by this column, we want to group rows in the following 3 groups: low sales (less than 10K), medium sales (from 10K to 100K), and high sales (100K and greater). The following code snippet shows a possible solution in iGrid:

```
// Grid setup code
...
// Add Sales column (same column key and title)
iGCol myCol = iGrid1.Cols.Add("Sales", "Sales");
// Use custom grouping in this column
myCol.CustomGrouping = true;
...

// The CustomGroupValue event handler
private void iGrid1_CustomGroupValue(
  object sender, iGCustomGroupValueEventArgs e)
{
  if (iGrid1.Cols.GetKey(e.ColIndex) == "Sales")
  {
    decimal mySales = (decimal)iGrid1.CellValues[e.RowIndex, e.ColIndex];
    if (mySales < 10000)
      e.Value = "Low Sales";
    else if (mySales < 100000)
      e.Value = "Medium Sales";
    else
      e.Value = "High Sales";
  }
}
```

When custom grouping is used for a column, iGrid sorts the group by custom group values regardless of the column sort type. This behavior is not always desirable. For example, in our sample we will see the groups in the following order — High Sales, Low Sales, Medium Sales — which is a bit unnatural. We may want to order the groups by sales size from smallest to highest keeping the original group names, i.e. Low Sales, Medium Sales, and High Sales. iGrid implements one more event related to custom grouping that will help us — this is **CustomGroupValueText**. This event is raised by iGrid when it needs to know the text representation of a custom group value. Let's use it to enhance our sample. The idea is to provide custom group values that will be used to order our groups in the required order in an event handler of the **CustomGroupValue** event, and then tell iGrid corresponding texts for group titles in an event handler of the **CustomGroupValueText** event.

We can use normal integer values like 0, 1, 2 as group values in our sample:

```
private void iGrid1_CustomGroupValue(
  object sender, iGCustomGroupValueEventArgs e)
{
  if (iGrid1.Cols.GetKey(e.ColIndex) == "Sales")
  {
    decimal mySales = (decimal)iGrid1.CellValues[e.RowIndex, e.ColIndex];
    if (mySales < 10000)
      e.Value = 0;
    else if (mySales < 100000)
      e.Value = 1;
    else
      e.Value = 2;
  }
}
```

Then provide the corresponding texts for them in the **CustomGroupValueText** event:

```
private void fGrid_CustomGroupValueText(
  object sender, iGCustomGroupValueTextEventArgs e)
{
  if (fGrid.Cols.GetKey(e.ColIndex) == "Sales")
  {
    switch ((int)e.Value)
    {
      case 0:
        e.Text = "Low Sales";
        break;
      case 1:
        e.Text = "Medium Sales";
        break;
      case 2:
        e.Text = "High Sales";
        break;
    }
  }
}
```

If you use the default column settings, when grouping by a column, its header is displayed in the group box, but the column itself disappears from the cell area. At the same time, we continue to see the values of the column cells in the created group rows. In the case of custom grouping, we will no longer see the values of the cells included in groups, although this can be very useful. For example, in our example with customer sales, we may want to see sales for each customer even when the grid is grouped by the Sales column and we have 3 groups with sale ranges. To keep the column visible in the cell area when the grid is grouped by it, simply set the **ShowWhenGrouped** property of the column to True. The grid setup code placed above would look like as follows with this improvement:

```
iGCol myCol = iGrid1.Cols.Add("Sales", "Sales");
myCol.CustomGrouping = true;
myCol.ShowWhenGrouped = true;
```

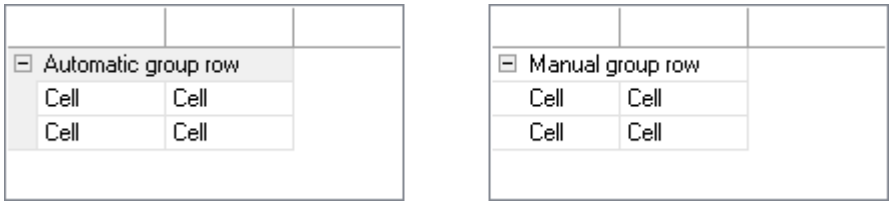
As a bonus, iGrid also sorts a column with custom grouping according to its sort type specified in the **SortType** property of the column. This feature complements the ability to view column cells when grouping and makes using iGrid even more convenient.

18.9. Manual Group Rows

Group rows in iGrid can be of two types: automatic and manual. Automatic group rows are fully managed by iGrid. They are created automatically when you group the grid by dragging a column header into the group box or when you call the **Group** method to group rows from code. They are also deleted by iGrid automatically on next grouping and the like.

Manual group rows, as their name suggests, are intended to be created manually. They are not processed by iGrid and are managed solely by the developer. They can be used for custom group headers, separators, or as special bars to display additional information for grid data.

The main visual difference between these two types of group rows is that the child rows of an automatic group row have a special bar in the level indent area connecting them visually to the corresponding group row, while this bar is absent for the child rows of a manual group row:



Manual group rows can be created only from code. When you create a row from code, you can specify its type using the **Type** property of the **iGRow** class. Set it to **ManualGroupRow** to create a manual group row.

To understand the difference between the two group row types better, look at the code used to create the first grid on the screenshot above:

```
iGrid1.Cols.Count = 3;
iGrid1.Cols[0].DefaultCellValue = "Automatic group row";
iGrid1.Cols[1].DefaultCellValue = "Cell";
iGrid1.Cols[2].DefaultCellValue = "Cell";
iGrid1.Rows.Count = 2;

iGrid1.GroupObject.Add(0);
iGrid1.Group();
```

Now compare it to the code used to create the grid with the manual group row:

```
iGrid1.Cols.Count = 2;
iGrid1.Cols[0].DefaultCellValue = "Cell";
iGrid1.Cols[1].DefaultCellValue = "Cell";

iGRow myRow;

myRow = iGrid1.Rows.Add();
myRow.Type = iGRowType.ManualGroupRow;
myRow.TreeButton = iGTreeButtonState.Visible;
myRow.RowTextCell.Value = "Manual group row";

myRow = iGrid1.Rows.Add();
myRow.Level = 1;
myRow = iGrid1.Rows.Add();
myRow.Level = 1;
```

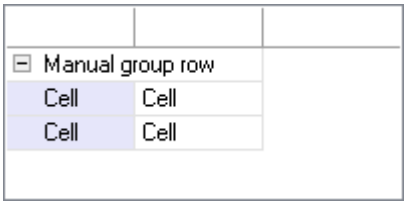
Pay attention to the fact that the plus/minus button is not displayed in a manual group row by default. You need to enable it explicitly with the **TreeButton** property of the row. The background color of a manual group row is also empty by default. You can specify it and other formatting options using the **CellStyle** property of the row, for example:

```
myRow.CellStyle.BackColor = Color.DarkSeaGreen;
```

Note that the levels of child rows of manual group rows must also be set explicitly with the **iGRow**.**Level** property. This will allow you to collapse a manual group row with all its child rows when the plus/minus button is clicked.

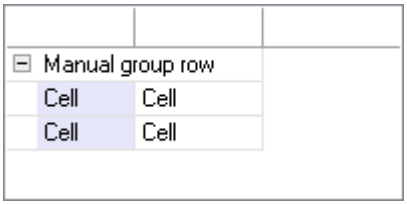
By default the level indent area inside normal cells of child rows is filled with the background color of these cells. We can visualize it better if we add the following statement to the code snippet above:

```
iGrid1.Cols[0].CellStyle.BackColor = Color.Lavender;
```



We can create a visual effect of hierarchy for child rows similar to the bar in the level indent area for automatic group rows if we disable coloring of the level indent area with the **ColorizeRowLevelIndent** property of iGrid:

```
iGrid1.ColorizeRowLevelIndent = false;
```



18.10. Child Row Visibility When Collapsing Group Rows

When a group row is collapsed, all its child group and normal rows become hidden. The **Visible** property of those child rows isn't changed at that. Another row property, **VisibleParentExpanded**, specifies whether a child row is visible in this scenario. To hide a row when one of its parent rows is collapsed, iGrid sets the row's **VisibleParentExpanded** property to False.

Thus, in the general case a row is visible only if the values of its all three visible-related properties — **Visible**, **VisibleFiltered**, and **VisibleParentExpanded** — equal True. This approach allows us to preserve the visibility of every row specified with the **Visible** property even if it is temporarily hidden on the screen when group rows are collapsed.

Remember about this when you add child rows to a collapsed group row from code: the **VisibleParentExpanded** property for such rows should be set to False. For instance, you can create a group row that is initially collapsed using code like this:

```
iGRow myParentRow = iGrid1.Rows.Add();
myParentRow.Type = iGRowType.AutoGroupRow;
myParentRow.TreeButton = iGTreeButtonState.Visible;
myParentRow.Expanded = false;
```

Then its hidden child rows should be created using a code snippet like the following one:

```
iGRow myChildRow = iGrid1.Rows.Add();
myChildRow.Level = 1;
myChildRow.VisibleParentExpanded = false;
```

18.11. Other iGrid Members Related to Grouping

- iGrid raises the **BeforeContentsGrouped** event before its rows are grouped (no matter in code or interactively with the group box). Immediately after the group rows have been created, iGrid raises the **AfterContentsGrouped** event. These events can be helpful if you implement special rows that should be added/changed/removed before and after grouping — for instance, custom subtotal rows you recalculate each time when iGrid is grouped. In this case the **BeforeContentsGrouped** event can be used to remove the subtotal rows you insert manually, and the **AfterContentsGrouped** event can be used to create new subtotal rows for the newly created groups.

- When the user is expanding or collapsing a group row, the **BeforeRowStateChanged** event is triggered. You can prohibit the change of the current expanded state by handling this event. If the expanded state has been changed, the **AfterRowStateChanged** event occurs. Note that these events work only when the row state is changed interactively but not from code.
- Most of the iGrid mouse events fired for column headers also work in the group box. Among them are **ColHdrClick**, **ColHdrStartDrag**, **ColHdrDragging**, **ColHdrEndDrag**. The event data of some of these events provide you with special fields allowing you to detect that the action is performed in the group box — such as the **ToGroupBox** field in the arguments of the **ColHdrEndDrag** event. These fields allow you to disable operations in the group box dynamically based on some conditions — for example, you can disable moving a column to the group box if a condition is false.
- The X-buttons in column headers have its own click event — **ColHdrXButtonClick**. You can use it to disable clicks on X-buttons. The **RequestColHdrElemControlToolTipText** event of iGrid can be used to provide tooltips for X-buttons.
- When you group iGrid by a column, the column is removed from the cell area, though its **Visible** property isn't changed. If you want this column to remain visible in the cell area in this case, set the column's **ShowWhenGrouped** property to True. To prohibit grouping by the column interactively, set the column's **AllowGrouping** property to False.
- If you have group rows in iGrid and sort it by columns not included into the grouping, iGrid sorts normal rows within their groups. This means that group rows do not change their position and none of the normal rows is moved out of its group. For more details, see Chapter Sorting.
- If the stay-sorted mode is on (see the **StaySorted** property of iGrid), you cannot add group rows to the grid programmatically or with the group box.
- The **AutoHeight** method of the **iGRow** class fits the height of a group row assuming the group row has an infinite width.

19. SUMMARIES

19.1. General Capabilities of the Summary Infrastructure

iGrid allows you to define summaries calculated for groups and total summaries displayed in footer cells. While each type of summary has its own unique capabilities, they all are built on the common principles outlined below.

Summary updates

By default, iGrid calculates and automatically updates summaries on any appropriate event. These are:

1. Adding or removing grid rows (using the **Count** property or methods like **Add**, **RemoveRange**, etc. of the **Rows** collection of **iGrid**).
2. Changing row visibility (setting the **Visible** or **VisibleFiltered** property of the **iGRow** object, but not while collapsing/expanding group rows or tree nodes).
3. Changing cell values (interactively or from code with the **Value** property of the **iGCell** object).
4. Populating iGrid from a data source using the **FillWithData** method.

iGrid implements the Boolean **AutoUpdateGroupSummaries** and **AutoUpdateFooterSummaries** properties you can use to turn off automatic calculation of group summaries and footer summaries respectively in all the cases listed above. This functionality can be used to speed up code execution if you definitely know that your changes to the grid in the cases listed above will not affect the calculation of specified summaries.

The automatic calculation of summaries is also temporarily turned off between calls to iGrid's **BeginUpdate** and **EndUpdate** methods. The fact is that if redrawing in iGrid is turned off after calling the **BeginUpdate** method, there is no sense in summary calculations after every change because this is not the final set of cell values for which summaries must be calculated. This built-in optimization leads to significant performance improvement.

iGrid raises the **GroupSummariesNeedUpdate** and **FooterSummariesNeedUpdate** events in every case when summaries for grid cells should be updated. These events can be helpful if you want to calculate specific summaries in your own algorithm. In fact, these events tell you when you need to recalculate your summaries if anything related to summaries has been changed in the grid. You can also use these event to perform some specific actions before summary calculation — for example, to initialize variables needed for filtered summary calculation in an event handler of the **IncludeCellInFooterSummary** event.

The **GroupSummariesNeedUpdate** and **FooterSummariesNeedUpdate** events are never raised if redrawing was turned off after calling the **BeginUpdate** method of iGrid. Note that if redrawing is on, these events are raised even when the built-in summary calculations were turned off by setting the **AutoUpdateGroupSummaries** or **AutoUpdateFooterSummaries** property to False.

Two other iGrid events related to summary calculations, **GroupSummariesUpdated** and **FooterSummariesUpdated**, are raised after the summary values have been updated by iGrid. These notification events can be used to retrieve the actual summary values or format them after calculation (for example, mark negative totals with red).

If necessary, you can always update the summary values using the **UpdateGroupSummaries** and **UpdateFooterSummaries** methods of iGrid.

Rows involved in summary calculation

iGrid calculates summary values only for visible rows. These are rows with the **Visible** and **VisibleFiltered** properties set to True.

iGrid also provides you with the ability to exclude individual cells from summary calculations. This is done with the help of its **IncludeCellInGroupSummary** and **IncludeCellInFooterSummary** events.

Processing of object cell values in numerical summary calculations

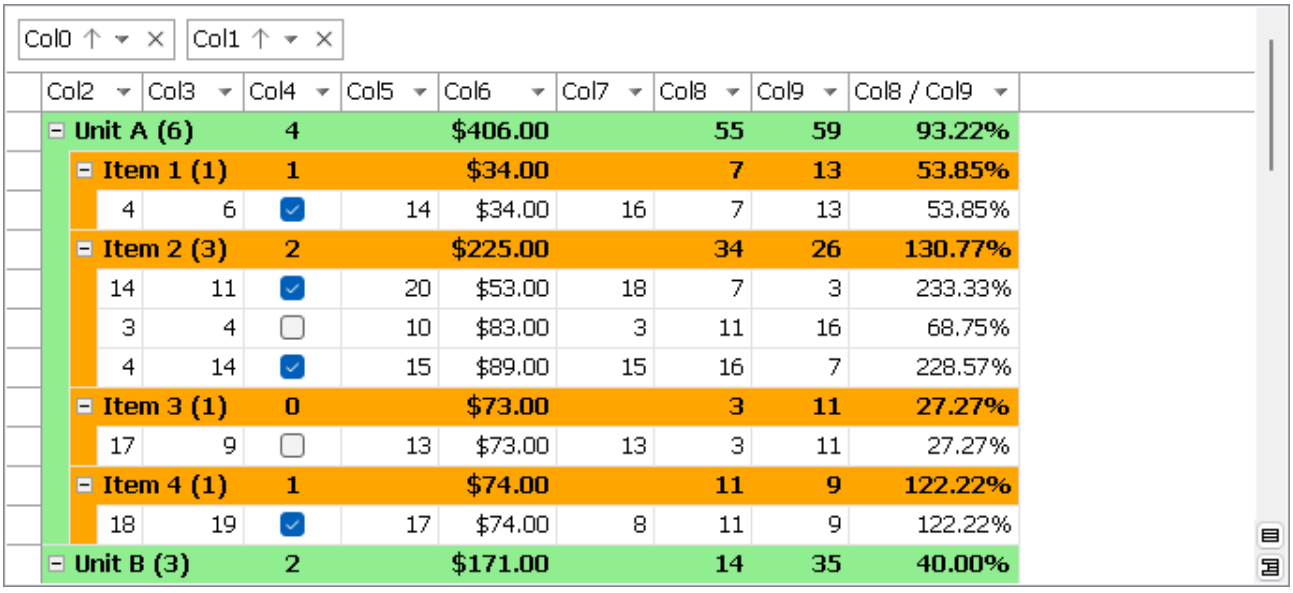
iGrid cell values are object values and can store more than just numeric values. When iGrid calculates a summary value, it tries to convert the cell values involved into the calculation to the corresponding values of the **System.Decimal** type. If the conversion of a cell value was successful, it is used in the calculation; if not, then it is simply skipped without throwing any error. This approach gives you some benefits, for example:

- Cells may contain empty strings or strings like "n/a" to indicate a missing value. This will not prevent iGrid from proper calculation of aggregate functions — such cells will be simply skipped during summary calculation.
- You can store numeric values as strings in the current regional numeric format. If iGrid can convert them to the corresponding **Decimal** representation with the platform's **Convert.ToDecimal** call, the summary values will also be calculated for string representation of numeric data.

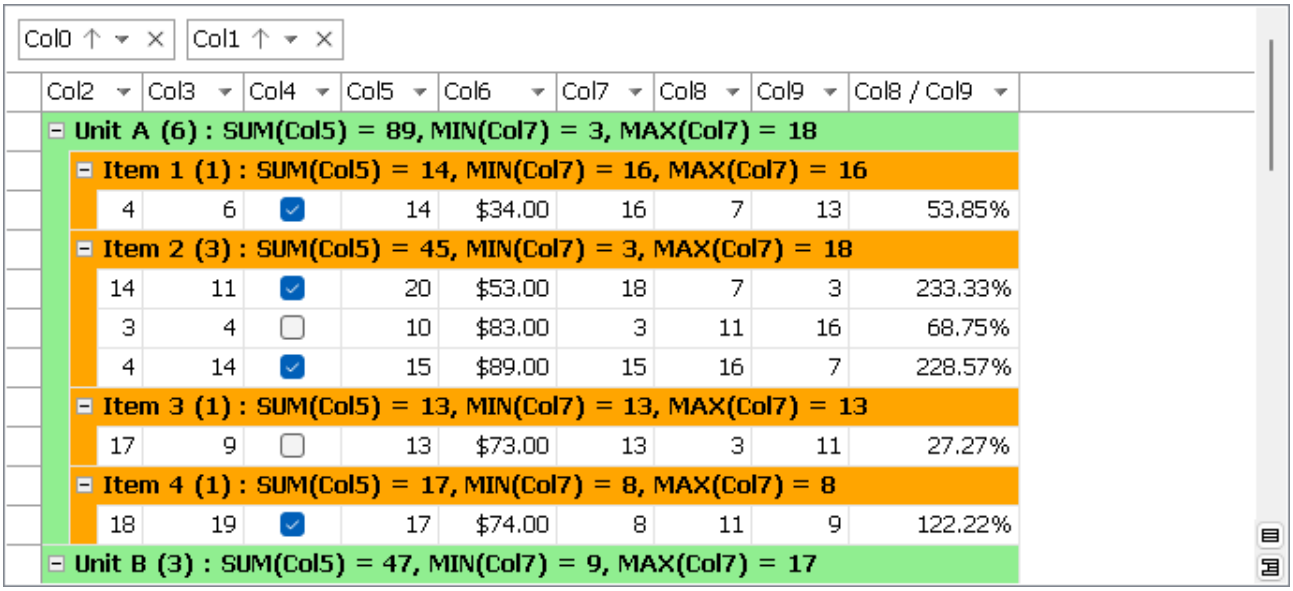
19.2. Group Summaries

19.2.1. Two Kinds of Group Summaries

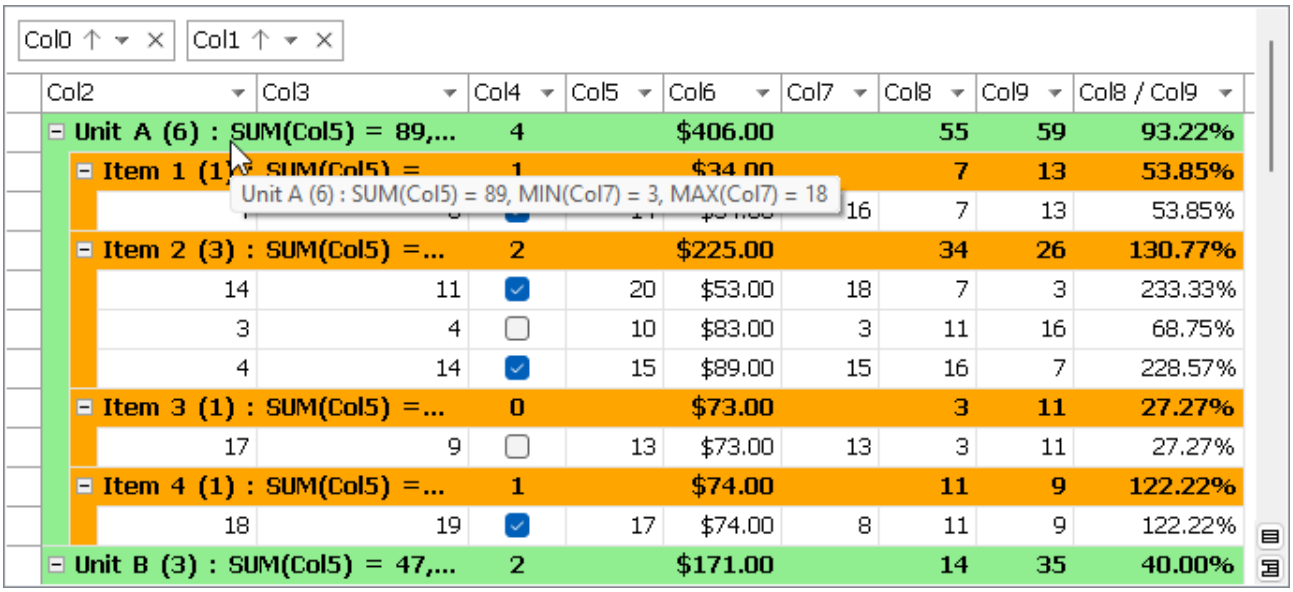
iGrid provides two kinds of group summaries: in-column group summaries and group header summaries. In-column group summaries are displayed in the cells of the columns for which they are calculated:



Group title summaries are stored in a group row and are automatically displayed as part of group title unless you have hidden them:



Both kinds of group summaries can be used within the same grid:



19.2.2. In-column Group Summaries

To define the in-column group summary for a column, use the **GroupSummaryType** property of the corresponding **iGCol** column object. This property holds a value from the **iGSummaryType** enumeration. If the **GroupSummaryType** property is set to a value other than **None** (the default value), iGrid calculates the corresponding aggregate function for the column cells and displays the result in group rows. Below is an example of how to show totals for the cells in the 5th column in group rows:

```
iGrid1.Cols[4].GroupSummaryType = iGSummaryType.Sum;
```

The summary value for a group is stored as the value of the corresponding cell in the group row. This means that you can read the calculated value like the traditional value of iGrid cells. For example, if you have configured iGrid to count the sum of the cell values in the 5th column and display these values in groups, you can retrieve the calculated sum for the very first group using the code below:

```
decimal mySum = (decimal)iGrid1.CellValues[0, 4];
```

Note that this approach allows you to calculate summary values for check box cells. In most cases check box states are stored as **Boolean**, **CheckState** or numeric values, and they can be treated as 1 for checked cells and 0 for unchecked cells. Thus, the **Sum** aggregate function for a column with check box cells will count the number of checked cells in the corresponding group.

If a group row displays an in-column group summary, iGrid outputs the summary value inside the group row using almost the same algorithm used to render data cells. This allows you to use almost all standard cell formatting features, such as color/font formatting, text alignment, etc. Column cell style inheritance also works for such cells, allowing you to apply the same format used for the column's data cells to the group summary values automatically.

The functionality of cells used to output in-column summaries inside group rows differs from data cells in the following aspects:

- The cell vertical grid line is not drawn but occupies the corresponding space as if it were visible.
- The **Type** and **TypeFlags** properties are not inherited from the column and row cell styles. The cell type is always considered text, and any type flags are not applied.

The foreground part of empty cells between cells with in-column summaries inside group rows is never rendered, but the background part is rendered like for data cells. This means you can specify their background color with the **BackColor** property of the corresponding cell or even use custom background drawing.

19.2.3. Group Title Summaries

In contrast to in-column group summaries, you can define group summaries designated for displaying in group titles. This is done with the help of the **GroupTitleSummaries** property of iGrid, which is a collection of summary definition objects. Each summary definition is an instance of the **iGSummaryDef** class with the following properties:

- **Key** — a unique string to reference this summary.
- **ColIndex** — the grid column for which summary is calculated.
- **SummaryType** — the summary type (aggregate function).
- **DisplayFormat** — the format string to get the text representation of summary value.

Every summary definition must have a key. Keys are case-sensitive and cannot be null. The target column and the summary type must also be specified. The display format is optional and is used only when you want to display the summary calculation result in group titles.

Group title summary definitions are created with the **Add** method of the **GroupTitleSummaries** collection. There are several overloaded versions of this method allowing you to specify the values of the aforementioned properties for the created **iGSummaryDef** objects. Below is an example demonstrating how to calculate totals for the column with the key "Data" and display them as "Amount = <value>" at the end of group titles:

```
iGrid1.GroupTitleSummaries.Add(
  "data_total", "Data", iGSummaryType.Sum, "Amount = {0:#,##.00}");
```

The first argument of the **Add** call above, the string "data_total", is the key that can be used to retrieve the values of this summary for groups.

Every overload of the **Add** method returns an **iGSummaryDef** object that represents the created summary definition. This object is an informational object allowing you only to read the properties of the summary. You cannot create **iGSummaryDef** objects in your code using the New operator because these objects are tightly linked to the internal iGrid data and can be managed correctly only by iGrid.

The **Add** methods create new summary definitions and add them to the end of the **GroupTitleSummaries** collection. This collection also implements the set of **Insert** methods with

the corresponding overloads like the **Add** method to create summary definitions and insert them before existing ones.

The **GroupTitleSummaries** collection implements other properties and methods similar to traditional members in any collection of objects, such as the **Count** property or the **Clear** method. You can also retrieve summary definitions using numeric indexes or string keys, enumerate them with a for-each loop and apply LINQ queries to the **GroupTitleSummaries** collection.

If the **DisplayFormat** property of a group title summary defined in the **GroupTitleSummaries** collection is not null or an empty string, the result of its calculation is automatically displayed at the end of group title. The string representation of the result is retrieved by applying the specified format to the raw summary value. iGrid constructs the resulting group title representation for all such summaries by combining their string representations into one string. The following 3 special string properties of the iGrid.GroupTitleSummaries object are used for building this resulting string:

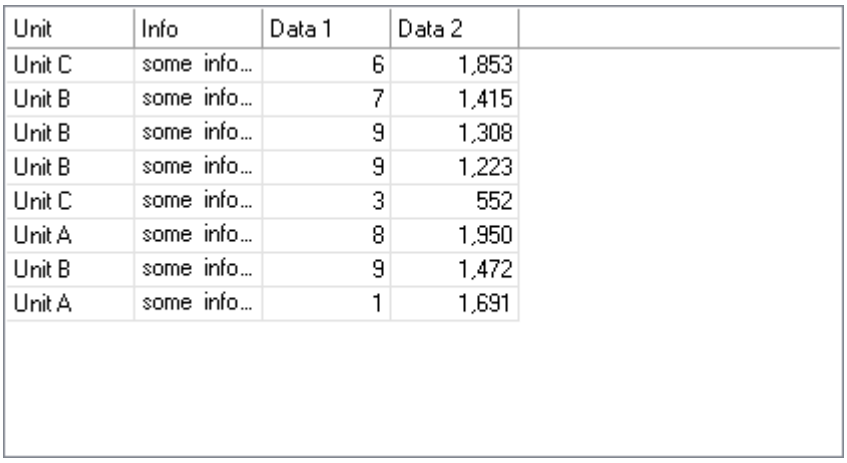
- **JoinValuesDelimiter** — the separator string between summary values (", " by default).
- **JoinValuesPrefix** — the string added before the resulting string (null by default).
- **JoinValuesSuffix** — the string added after the resulting string (null by default).

The calculated values of group title summaries are stored in the **SummaryValues** sub-property of the **GroupInfo** object property of an **iGRow** row object. **SummaryValues** is an instance of the .NET **OrderedDictionary** class in which you can access items by their numeric indexes or string keys. These numeric indexes correspond to the indexes of summary definitions in the iGrid **GroupTitleSummaries** collection. The strings keys are the keys of the summary definitions, and this is one of the reasons why they must be defined while creating summary definitions with the **Add** or **Insert** methods of the **GroupTitleSummaries** collection. Below is an example showing how to display the summary value for the "data_total" summary definition we created above for the first group row:

```
private void buttonDisplayDataAmount_Click(object sender, EventArgs e)
{
  object dataTotal = iGrid1.Rows[0].GroupInfo.SummaryValues["col5sum"];
  MessageBox.Show($"Total amount of data: {dataTotal}");
}
```

19.2.4. Examples of Group Summaries

This topic will demonstrate the main features of iGrid group summaries in action. We will use the following sample grid for the demonstration:



Unit	Info	Data 1	Data 2
Unit C	some info...	6	1,853
Unit B	some info...	7	1,415
Unit B	some info...	9	1,308
Unit B	some info...	9	1,223
Unit C	some info...	3	552
Unit A	some info...	8	1,950
Unit B	some info...	9	1,472
Unit A	some info...	1	1,691

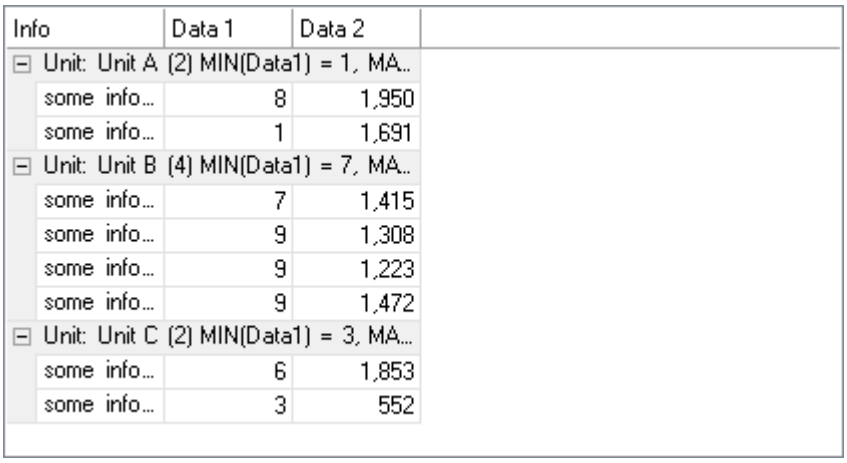
It was created with the following code:

```
iGrid1.Cols.Add("unit", "Unit");
iGrid1.Cols.Add("info", "Info");
iGrid1.Cols.Add("data1", "Data 1");
iGrid1.Cols.Add("data2", "Data 2");

iGrid1.Cols["data1"].CellStyle.TextAlign = iGContentAlignment.MiddleRight;
iGrid1.Cols["data2"].CellStyle.TextAlign = iGContentAlignment.MiddleRight;
iGrid1.Cols["data2"].CellStyle.FormatString = "{0:N0}";

iGrid1.Rows.Count = 8;

string[] units = { "Unit A", "Unit B", "Unit C" };
Random random = new Random(100);

foreach (iGRow row in iGrid1.Rows)
{
  row.Cells["unit"].Value = units[random.Next(units.Length)];
  row.Cells["info"].Value = "some info " + random.Next(100, 1000);
  row.Cells["data1"].Value = random.Next(10);
  row.Cells["data2"].Value = random.Next(500, 2000);
}
```

Let's suppose we want to group this grid by the values of the Unit column and need some summaries for every unit — the minimal and maximal values stored in the column "Data 1", and the sum of values stored in the column "Data 2". The simplest way to implement this task is the following code:

```
iGrid1.GroupTitleSummaries.Add(
  "data1min", "data1", iGSummaryType.Min, "MIN(Data1) = {0}");
iGrid1.GroupTitleSummaries.Add(
  "data1max", "data1", iGSummaryType.Max, "MAX(Data1) = {0}");
iGrid1.GroupTitleSummaries.Add(
  "data2sum", "data2", iGSummaryType.Sum, "SUM(Data2) = {0:N0}");

iGrid1.GroupObject.Add("unit");
iGrid1.Group();
```

After adding this code to the setup code of our grid and launching it we will see the following result:

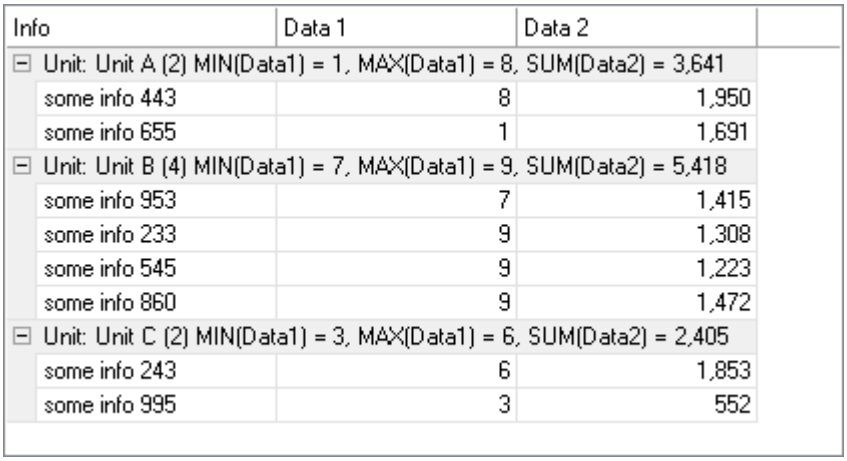


Info	Data 1	Data 2
⊟ Unit: Unit A (2) MIN(Data1) = 1, MA...		
some info...	8	1,950
some info...	1	1,691
⊟ Unit: Unit B (4) MIN(Data1) = 7, MA...		
some info...	7	1,415
some info...	9	1,308
some info...	9	1,223
some info...	9	1,472
⊟ Unit: Unit C (2) MIN(Data1) = 3, MA...		
some info...	6	1,853
some info...	3	552

The default widths of columns are not enough to display group titles with summaries without clipping. We can fix this problem by calling the **AutoWidth** method for the row text column that stores group titles this way:

```
iGrid1.RowTextCol.AutoWidth(iGRowTextKinds.GroupRow);
```

Now the result looks much better:



Info	Data 1	Data 2
⊟ Unit: Unit A (2) MIN(Data1) = 1, MAX(Data1) = 8, SUM(Data2) = 3,641		
some info 443	8	1,950
some info 655	1	1,691
⊟ Unit: Unit B (4) MIN(Data1) = 7, MAX(Data1) = 9, SUM(Data2) = 5,418		
some info 953	7	1,415
some info 233	9	1,308
some info 545	9	1,223
some info 860	9	1,472
⊟ Unit: Unit C (2) MIN(Data1) = 3, MAX(Data1) = 6, SUM(Data2) = 2,405		
some info 243	6	1,853
some info 995	3	552

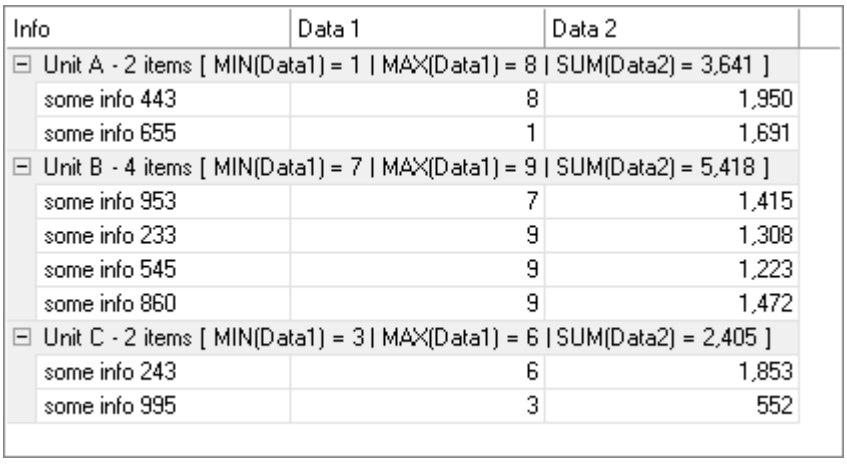
However, we can improve it even further by using the built-in configuration options of iGrid. For example, we can remove the unnecessary column title prefix before each unit name and display the number of items with a hyphen like this:

```
iGrid1.GroupTitleOptions.AddColTitlePrefix = false;
iGrid1.GroupTitleOptions.ItemCountFormatString = "- {0} items";
```

For better perception, we can display the summary values in square brackets and separate them with a non-standard separator:

```
iGrid1.GroupTitleSummaries.JoinValuesPrefix = "[ ";
iGrid1.GroupTitleSummaries.JoinValuesDelimiter = " | ";
iGrid1.GroupTitleSummaries.JoinValuesSuffix = " ]";
```

The result of all our improvements is shown in the screenshot below:

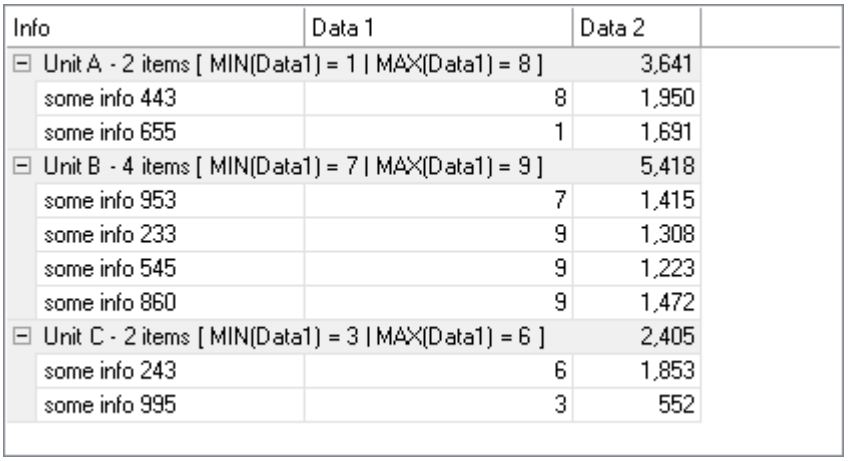


Info	Data 1	Data 2
⊟ Unit A - 2 items [MIN(Data1) = 1 | MAX(Data1) = 8 | SUM(Data2) = 3,641]		
some info 443	8	1,950
some info 655	1	1,691
⊟ Unit B - 4 items [MIN(Data1) = 7 | MAX(Data1) = 9 | SUM(Data2) = 5,418]		
some info 953	7	1,415
some info 233	9	1,308
some info 545	9	1,223
some info 860	9	1,472
⊟ Unit C - 2 items [MIN(Data1) = 3 | MAX(Data1) = 6 | SUM(Data2) = 2,405]		
some info 243	6	1,853
some info 995	3	552

To demonstrate in-column group summaries and their use with group title summaries in the same grid, let's calculate the total for the column "Data 2" using the **iGCol.GroupSummaryType** property instead of the **GroupTitleSummaries** property of iGrid. The code snippet defining group summaries will now look like this:

```
iGrid1.GroupTitleSummaries.Add(
  "data1min", "data1", iGSummaryType.Min, "MIN(Data1) = {0}");
iGrid1.GroupTitleSummaries.Add(
  "data1max", "data1", iGSummaryType.Max, "MAX(Data1) = {0}");
iGrid1.Cols["data2"].GroupSummaryType = iGSummaryType.Sum;
```

The result of this determination of summaries is as follows:



Info	Data 1	Data 2
⊟ Unit A - 2 items [MIN(Data1) = 1 | MAX(Data1) = 8]		3,641
some info 443	8	1,950
some info 655	1	1,691
⊟ Unit B - 4 items [MIN(Data1) = 7 | MAX(Data1) = 9]		5,418
some info 953	7	1,415
some info 233	9	1,308
some info 545	9	1,223
some info 860	9	1,472
⊟ Unit C - 2 items [MIN(Data1) = 3 | MAX(Data1) = 6]		2,405
some info 243	6	1,853
some info 995	3	552

Note that the totals in the column "Data 2" automatically use the same display format as the data in normal data cells.

19.3. Footer Summaries

19.3.1. Aggregate Functions in Footer Cells

Automatic calculation of summaries in footer cells

Footer cells can be configured to automatically calculate and display column summaries. Every footer cell is represented with an instance of the **iGFooterCell** class that implements the **SummaryType** property used for this purpose. This property of the **iGSummaryType** enumeration type allows you to specify one of the 5 available aggregate functions: Sum, Average, Minimum, Maximum, or Count. Below is an example how to calculate and display the sum of cell values in the 6th column in the footer:

```
iGrid1.Footer.Cells[0, 5].SummaryType = iGSummaryType.Sum;
```

By default the **SummaryType** property for all footer cells is set to **None**. If at least one footer cell has a non-default value in the **SummaryType** property, iGrid calculates the corresponding aggregate function(s) automatically on any appropriate event as it is described in the General Capabilities of the Summary Infrastructure topic.

Retrieving calculated summary values from code

The result of the calculation of an aggregate function is stored in two properties of the footer cell — the traditional **Value** property and the dedicated **Total** property. The **Value** property is the traditional way to retrieve the value the user sees on the screen. However, this property has the **Object** type. This means that if you need to use the aggregate function result in further mathematical calculations or comparison conditions, first you will need to check whether this value is not null (Nothing in VB) and then convert the boxed numeric value to the appropriate numeric type. That's why iGrid provides another property to retrieve the calculated summary values — **iGFooterCell.Total**. This property returns the calculated result as a **Decimal** value you can use "as is" in calculations/comparisons.

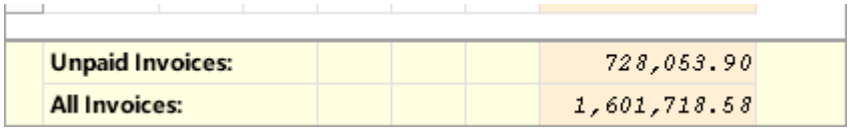
Note that the **Value** is a read-write property, whereas **Total** is a read-only one. The former property can be rewritten with any other value if required, but the latter property will still contain the calculated result until the next summary calculation.

19.3.2. Several Aggregate Functions for the Same Column

If you have several rows in the footer, you can define more than one aggregate function for the same column. For example, the first footer cell in the 4th column can display the row counter and the second footer cell can display the sum:

```
iGrid1.Footer.Cells[0, 3].SummaryType = iGSummaryType.Count;
iGrid1.Footer.Cells[1, 3].SummaryType = iGSummaryType.Sum;
```

iGrid extends this idea even more and allows you to define the same aggregate function for the same column several times, but filter rows included into calculations of a particular aggregate function. A practical example would be a list of invoices in your grid, where you calculate two sums: the sum of all invoices and the sum of unpaid invoices. It may look like the following picture:



Row filtering for summary calculations is performed with the **IncludeCellInFooterSummary** event of **iGrid**. The event arguments indicate which cell (**RowIndex**, **ColIndex**) is about to be used for the summary calculation in a footer row (**FooterRowIndex**) and whether to do that (**DoDefault**). The footer section on the above picture was created with the help of the **IncludeCellInFooterSummary** event:

```
iGrid1.Footer.Rows.Count = 2;
iGrid1.Footer.Cells[0, 0].Value = "Unpaid Invoices:";
iGrid1.Footer.Cells[0, 0].SpanCols = 3;
iGrid1.Footer.Cells[1, 0].Value = "All Invoices:";
iGrid1.Footer.Cells[1, 0].SpanCols = 3;
iGrid1.Footer.Cells[0, 6].SummaryType = iGSummaryType.Sum;
iGrid1.Footer.Cells[1, 6].SummaryType = iGSummaryType.Sum;
iGrid1.IncludeCellInFooterSummary += iGrid1_IncludeCellInFooterSummary;

void iGrid1_IncludeCellInFooterSummary(
  object sender, iGIncludeCellInFooterSummaryEventArgs e)
{
   if (e.FooterRowIndex == 0)
   {
      e.DoDefault = (bool)iGrid1.Cells[e.RowIndex, "unpaid"].Value;
   }
}
```

We also imply that all values in the last column (both cell values and footer values) use the same format string, alignment and background color defined in the column's cell style:

```
iGCellStyle colCellStyle = iGrid1.Cols[6].CellStyle;
colCellStyle.BackColor = Color.PapayaWhip;
colCellStyle.TextAlign = iGContentAlignment.MiddleRight;
colCellStyle.FormatString = "{0:#,##.00}";
```

19.4. Custom Group and Footer Summaries

To implement custom summaries in group rows and footer cells, the **iGSummaryType** enumeration provides the **Custom** member. If you set the **GroupSummaryType** property of a column object (**iGCol**) or the **SummaryType** property of a group title summary definition object (**iGSummaryDef**) to **Custom**, iGrid raises the **CustomGroupSummary** event so that you can provide a custom summary value for a group row in all appropriate cases — creation of an automatic group row during grouping, group summary updates after editing a cell, etc.

The **CustomGroupSummary** event fires after all defined group summaries other than **Custom** have been automatically calculated. This allows you to calculate custom summary values based on the summary values calculated by iGrid. As an example, let's consider a grid in which sums of cell values by groups are calculated for columns with indexes 4 and 5. We need to calculate a custom group summary showing the ratio of these two values and display it in column with the index 6.

The definition of group summaries for our grid should look as follows:

```
iGrid1.Cols[4].GroupSummaryType = iGSummaryType.Sum;
iGrid1.Cols[5].GroupSummaryType = iGSummaryType.Sum;
iGrid1.Cols[6].GroupSummaryType = iGSummaryType.Custom;
```

The event handler of the **CustomGroupSummary** event implementing our task may look like this:

```
private void iGrid1_CustomGroupSummary(
  object sender, iGCustomGroupSummaryEventArgs e)
{
  If (e.ColIndex == 6)
  {
    decimal myValue1 = Convert.ToDecimal(iGrid1.CellValues[e.RowIndex, 4]);
    decimal myValue2 = Convert.ToDecimal(iGrid1.CellValues[e.RowIndex, 5]);

    if (myValue2 != 0)
      e.Value = myValue1 / myValue2;
  }
}
```

The similar event for footer cells is **CustomFooterSummary**. It is raised for footer cells with the **SummaryType** property set to **Custom**. This event is also raised after all footer summaries other than Custom were automatically calculated by iGrid, allowing you to calculate custom footer summary values based on the summary values calculated by iGrid.

Let's continue with our example. If we want to display total summaries for columns 4, 5, and 6 for the grid we described above, we will define the footer summaries as follows:

```
iGrid1.Footer.Cells[0, 4].SummaryType = iGSummaryType.Sum;
iGrid1.Footer.Cells[0, 5].SummaryType = iGSummaryType.Sum;
iGrid1.Footer.Cells[0, 6].SummaryType = iGSummaryType.Custom;
```

Then the following event handler of the **CustomFooterSummary** event will implement what we need:

```
private void iGrid1_CustomFooterSummary(
  object sender, iGCustomFooterSummaryEventArgs e)
{
  If (e.ColIndex == 6)
  {
    decimal myValue1 = iGrid1.Footer.Cells[e.RowIndex, 4].Total;
    decimal myValue2 = iGrid1.Footer.Cells[e.RowIndex, 5].Total;

    if (myValue2 != 0)
      e.Value = myValue1 / myValue2;
  }
}
```

The arguments of the **CustomGroupSummary** event are represented with an instance of the **iGCustomGroupSummaryEventArgs** class. Its **RowIndex** and **ColIndex** fields allow you to know the index of the group row and the index of the grid column respectively for which the custom summary is calculated. The **SummaryKey** field of the event arguments contains the key of the group title summary or null if the event is raised for a custom in-column group summary. The read-write **Value** field is used to tell iGrid the custom summary value calculated by the developer.

The arguments of the **CustomFooterSummary** event are provided with an instance of the similar **iGCustomFooterSummaryEventArgs** class, except it does not have the **SummaryKey** field used only for group summaries.

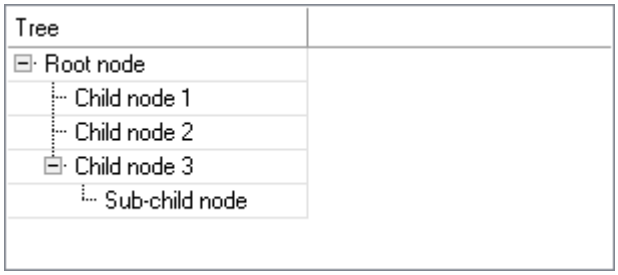
20. TREE GRID FUNCTIONALITY

20.1. Tree Grid Basics

The main idea of iGrid.NET is to be a container for tabular data, but you may need a hierarchical data representation if your rows are related to each other. One of the features you can use for that is grouping. It is based on the concept of row levels which is generally used for group rows. Another use of the row level engine for rows with normal cells is multi-column tree representation of your data, also known as tree grids.

All you need to do to create a tree in iGrid is to specify the hierarchy level for each row using its **Level** property and tell the row to display the plus/minus button with the **TreeButton** property of the row. The **TreeButton** property should be set to the **Visible** or **Hidden** state depending on whether the current row has child items or not. The default value of the **TreeButton** property is **Absent**. It seems the same as **Hidden**, but not: if the tree button is hidden, there is a place for it on the screen. You should use **Hidden** if you create a tree because this causes iGrid to draw proper indents for your tree items.

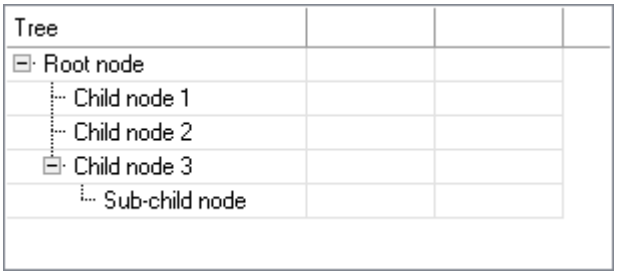
Here is a quick example to understand how it works. Let's create a tree like on the screenshot below:



The code is the following:

```
// Var to reference the last added row
iGRow myRow;

// Create one column for our tree
iGrid1.Cols.Add("Tree", 150);

// Add the first root node
myRow = iGrid1.Rows.Add();
myRow.Level = 0;
myRow.TreeButton = iGTreeButtonState.Visible;
myRow.Cells[0].Value = "Root node";

// Add two child nodes to the root without nested items
myRow = iGrid1.Rows.Add();
myRow.Level = 1;
myRow.TreeButton = iGTreeButtonState.Hidden;
myRow.Cells[0].Value = "Child node 1";

myRow = iGrid1.Rows.Add();
myRow.Level = 1;
myRow.TreeButton = iGTreeButtonState.Hidden;
myRow.Cells[0].Value = "Child node 2";

// Add one more our root's child node with a nested item
myRow = iGrid1.Rows.Add();
myRow.Level = 1;
myRow.TreeButton = iGTreeButtonState.Visible;
myRow.Cells[0].Value = "Child node 3";

myRow = iGrid1.Rows.Add();
myRow.Level = 2;
myRow.TreeButton = iGTreeButtonState.Hidden;
myRow.Cells[0].Value = "Sub-child node";

// Show tree lines for better structure indication
iGrid1.TreeLines.Visible = true;
```

If we added two more columns to our grid while defining columns, our grid would look like a multi-column tree grid in which we could set cell values and other formatting properties for the cells in the second and third columns:

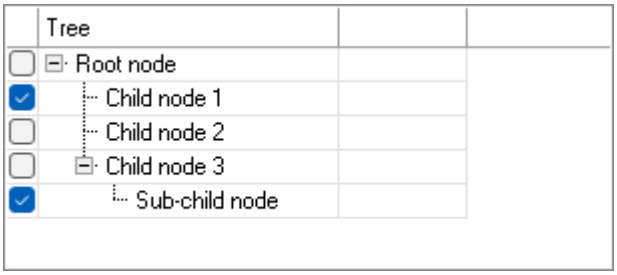


Tree nodes are expanded by default — which corresponds to the default value of the **Expanded** property of a new row. If you want to create a tree node in the collapsed state, set the **Expanded** property of the corresponding row to False. If this tree node will have child nodes, they must be also set as hidden after collapsing their parent by setting their **VisibleParentExpanded** property to False. You can find more information about this in the Child Nodes Visibility topic in this section.

If you do not specify the tree column explicitly, iGrid displays your tree in the first visible column and automatically "applies" your tree structure to any column the user can place on the first visible

place. You may need to prohibit the movement of the first column if you do not need this effect in your application.

It is important to note that any column of iGrid can be designated as the tree column, not just the first one. To display your tree in a specific column and have the ability to move it to any place by dragging its column header, assign a reference to the tree column to the **TreeCol** property. This allows you to display extra info or controls like check boxes before your tree nodes, for example:



This tree grid with check boxes was created by modifying the code snippet above. The core modification was in the second statement defining the column set:

```
iGrid1.Cols.Add();
iGrid1.Cols[0].CellStyle.Type = iGCellType.Check;
iGrid1.Cols.Add("Tree", 150);
iGrid1.TreeCol = iGrid1.Cols[1];
iGrid1.Cols.Add();
```

We also automatically adjusted the width of the first column with check boxes with the **iGCol**.**AutoWidth** method after populating the grid.

20.2. Customizing Tree Grids

iGrid allows you to adjust the appearance of tree lines and change their visibility. These settings are done through the **TreeLines** object property of the object **iGTreeLines** data type with the following properties:

- **Color** of the **Color** data type;
- **DashStyle** of the **DashStyle** data type;
- **ShowRootLines** (boolean);
- **Visible** (boolean).

The **Visible** property is used to show/hide the tree lines. The **ShowRootLines** property specifies whether the tree lines between the items on the first level are drawn (note that tree buttons are not hidden in this case — only the tree lines are hidden). The default values for these properties are True.

The **Color** and **DashStyle** properties control the look of the tree lines, their default values are **WindowText** and **Dot** respectively.

Remember that iGrid can draw the tree lines properly only if you prepare its rows for the tree view by setting their **TreeButton** properties to a non-default value which implies that there is a place for the tree lines (**Visible** or **Hidden**).

The size of the indent of one hierarchy level can be changed with the **LevelIndentOverride** property of iGrid. The **ColorizeRowLevelIndent** property of iGrid specifies whether the row level indent area is filled with the effective background color of the cell.

20.3. Child Nodes Visibility

When you collapse a tree node, all its child nodes become hidden. Like in the case of group rows, the **Visible** property of child rows isn't changed in this case. Another property that controls the visibility of an iGrid row when one of its parents is collapsed, **VisibleParentExpanded**, is set to False for this.

The same infrastructure based on row levels that works for group rows is in effect for tree nodes as well. You can find out more about it in the Child Row Visibility When Collapsing Group Rows topic.

Just remember that you need to set the **VisibleParentExpanded** property to False when you add child tree nodes to a collapsed node like in the following code:

```
iGRow myParentNode = iGrid1.Rows.Add();
myParentNode.Cells[0].Value = "Parent";
myParentNode.TreeButton = iGTreeButtonState.Visible;
myParentNode.Expanded = false;

iGRow myChildNode = iGrid1.Rows.Add();
myChildNode.Cells[0].Value = "Child";
myChildNode.Level = 1;
myChildNode.TreeButton = iGTreeButtonState.Hidden;
myChildNode.VisibleParentExpanded = false;
```

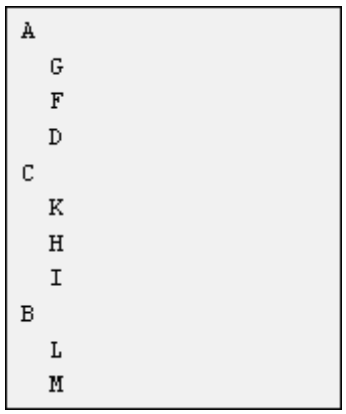
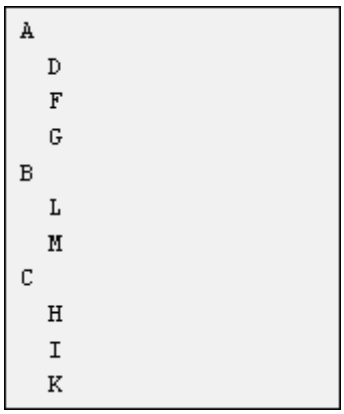
When the user tries to expand or collapse a tree node, the **BeforeRowStateChanged** event is triggered. If you process this event, you can prohibit the change of the current expanded state. If the expanded state has been changed, the **AfterRowStateChanged** occurs. Note that these events work only when the row state is changed interactively but not from code.

To collapse or expand a tree node from code, toggle the value of the Boolean **Expanded** property of the corresponding row.

20.4. Sorting Tree Grids

iGrid provides you with special sorting which allows you to sort rows taking into accounts theirs levels. In this case iGrid preserves the hierarchical structure of the rows and sort each hierarchical level independently. To activate this mode, use the **SortByLevels** property of iGrid.

The following pictures illustrate how hierarchical sorting works:

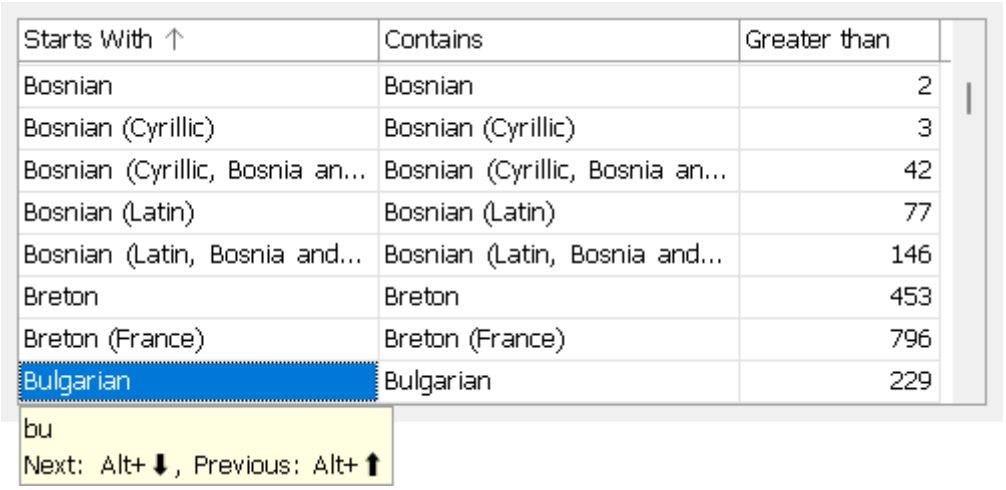
BEFORE SORTING BY LEVELS:	**AFTER SORTING BY LEVELS:**
	

21. SEARCH-AS-TYPE FUNCTIONALITY

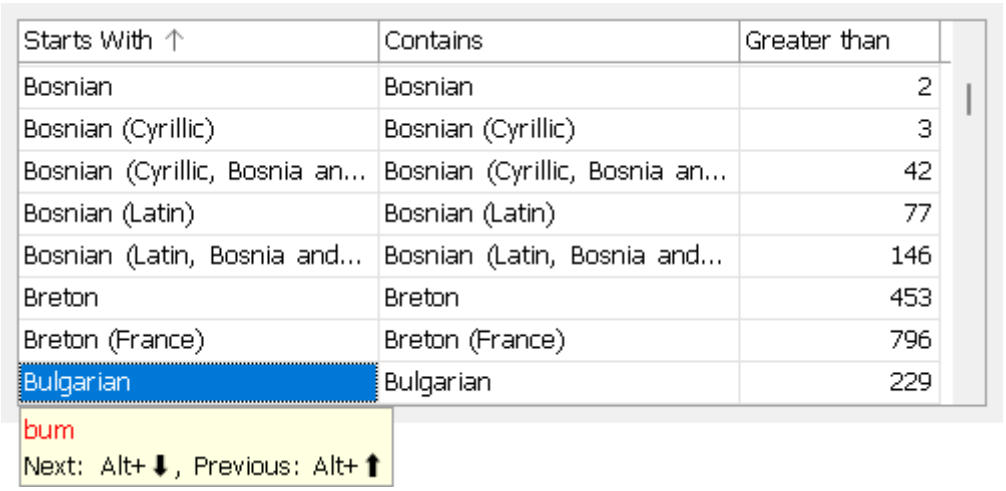
21.1. Search-as-Type Basics

The iGrid search-as-type functionality, also known as incremental search, allows the user to find the necessary cell containing a substring by typing this substring.

When the search-as-type functionality is active, iGrid displays a special search window when the user starts typing while iGrid has input focus. The search window displays the entered search string:



When the user enters a next character to search, it is added to the end of the search string. If no matches are found after adding the last typed character, the search string is displayed in red:



The following key combinations can be used while the search window is displayed:

- ALT+DOWN ARROW — go to the next matching cell.
- ALT+UP ARROW — go to the previous matching cell.
- BACKSPACE — remove the last character from the search string.
- ESC — cancel search.

Search-as-type works in two modes: seek and filter. In seek mode, iGrid moves the current cell to the position that matches the search criteria. In filter mode, iGrid leaves visible only those rows that meet the search criteria.

To control the search-as-type functionality from code, use the **SearchAsType** object property of iGrid. It returns an instance of the **iGSearchAsType** class whose properties are used to configure the functionality.

Search-as-type can be enabled only from code. To do that, choose the search mode (**Seek** or **Filter**) in the **Mode** property of the **SearchAsType** object. The default value of this property is **None**, which leads to editing when the user starts typing in iGrid cells. For example, you can activate seek in the current column in an iGrid with default settings using the following statement:

```
iGrid1.SearchAsType.Mode = iGSearchAsTypeMode.Seek;
```

Another property of the **SearchAsType** object frequently used to configure the search-as-type functionality is **SearchCol**. It allows you to specify the column used for search. By default this property is set to null (Nothing in VB) and iGrid searches for the entered string in the current column. But if you assign a column object to this property, iGrid will search in the specified column regardless of the column containing the current cell. Below is an example how to activate filtering by the first column:

```
iGrid1.SearchAsType.SearchCol = iGrid1.Cols[0];
iGrid1.SearchAsType.Mode = iGSearchAsTypeMode.Filter;
```

The criteria used to determine the cells that match the search string is specified with the **MatchRule** property of the **SearchAsType** object. This property accepts one of the following values:

- **StartsWith** — the cell text should start with the entered text.
- **Contains** — the cell text should contain the entered text.
- **Custom** — the match rule is determined by the developer in an event handler of the **SearchAsTypeCustomCompare** event (see the Custom Match Rules topic).

The default match rule is **StartsWith**.

The search-as-type functionality ignores all rows you made invisible by setting their **iGRow**.**Visible** to False from code. If search-as-type works in filter mode, it changes the **iGRow**.**VisibleFiltered** property to make a row visible or not according to the current filter. Read also the Row Visibility System topic to find out more about the iGrid row visibility system.

If the search-as-type works in filter mode, iGrid raises the **SearchAsTypeRowSetChanged** event to inform you about row visibility changes after a new filter criteria has been applied.

21.2. Adjusting Search-as-Type Functionality

In most cases the same match rule is used for all columns in the grid, but you can change this behavior by handling the **RequestSearchAsTypeMatchRule** event. It is raised every time when search-as-type starts. The **MatchRule** property of the event arguments allows you to specify a match rule for the column dynamically. The following example shows how to apply different match rules to different columns:

```
private void fGrid_RequestSearchAsTypeMatchRule(
  object sender, iGRequestSearchAsTypeMatchRuleEventArgs e)
{
  switch (e.ColIndex)
  {
    case 0:
      e.MatchRule = iGMatchRule.StartsWith;
      break;
    case 1:
      e.MatchRule = iGMatchRule.Contains;
      break;
    case 2:
      e.MatchRule = iGMatchRule.Custom;
      break;
  }
}
```

The **StartFromCurRow** property of the **SearchAsType** object allows you to specify whether to start search from the current row or from the beginning of the grid. The **LoopSearch** property of the **SearchAsType** object specifies whether to continue search from the first row of the grid when the last row is reached.

By default, the search window displays keyboard shortcuts used to move to the next and previous records matching the search criteria (ALT+DOWN ARROW and ALT+UP ARROW respectively). The **DisplayKeyboardHint** property of the **SearchAsType** object allows you to hide this hint. The **DisplaySearchText** property of the **SearchAsType** object can be used to hide the search text. If both properties are set to False, the search window is not displayed at all.

The "Next" and "Previous" strings in the keyboard hint can be changed or translated into another language with the **SearchWindowLabelNext** and **SearchWindowLabelPrev** properties of the **UIStrings** object property of iGrid.

iGrid uses the string comparison rules specific for the current culture (the culture of the current UI thread) while performing search-as-type operations. You can configure iGrid to use culture-independent string comparison rules with the **SearchAsTypeStringComparison** property of **iGrid**. For more information, read the Common Settings for Search-as-Type Tools topic.

21.3. Custom Match Rules

If the **Custom** value is specified in the **Mode** property of the **SearchAsType** object, iGrid raises the **SearchAsTypeCustomCompare** event when it needs to determine whether a cell matches the search criteria. If it is so, the **Match** parameter of the event should be set to True.

Below is an example showing how to implement custom searching. In the example we search cell values that are greater than the entered number:

```
// The last search text. This field is used in order
// to prevent parsing the search text for every comparison.
private string fLastSearchText;

// Stores the last search number. This field is used in order
// to prevent parsing the search text for every comparison.
private int fLastSearchNumber;

// Indicates whether the last search text cannot be converted
// to a number.
private bool fLastSearchNumberIsIncorrect;

// Compares the values in the third column with the search text.
private void fGrid_SearchAsTypeCustomCompare(
  object sender, iGSearchAsTypeCustomCompareEventArgs e)
{
  // If the last search text is not equal to the current,
  // parse it and memorize
  if (e.SearchText != fLastSearchText)
  {
    fLastSearchText = e.SearchText;
    try
    {
      fLastSearchNumber = int.Parse(e.SearchText);
      fLastSearchNumberIsIncorrect = false;
    }
    catch
    {
      fLastSearchNumberIsIncorrect = true;
    }
  }

  // If the current search text cannot be converted into a number,
  // return false
  if (fLastSearchNumberIsIncorrect)
  {
    e.Match = false;
    return;
  }

  // If the current cell does not contain a value, return false
  object myCellValue = fGrid.Cells[e.RowIndex, e.ColIndex].Value;
  if (myCellValue == null)
  {
    e.Match = false;
    return;
  }

  e.Match = Convert.ToDecimal(myCellValue) > fLastSearchNumber;
}
```

21.4. Using Search-as-Type Functionality from Code

As a rule, the search-as-type functionality is used interactively. You set the required options when iGrid is initialized, and iGrid does the rest by processing user keyboard input. But the features of the

iGrid search-as-type functionality can also be used from code — for example, to filter iGrid. Below you see a code snippet demonstrating how to configure the **SearchAsType** object to filter iGrid by substrings of the cell texts in the fifth column:

```
iGrid1.SearchAsType.SearchCol = fGrid.Cols[4];

iGrid1.SearchAsType.Mode = iGSearchAsTypeMode.Filter;
iGrid1.SearchAsType.MatchRule = iGMatchRule.Contains;

iGrid1.SearchAsType.DisplaySearchText = false;
iGrid1.SearchAsType.DisplayKeyboardHint = false;
iGrid1.SearchAsType.AutoCancel = false;
iGrid1.SearchAsType.FilterKeepCurRow = true;
```

The last 4 operators show the typical setting of the **SearchAsType** object when the search is done from code. Let's consider them.

The **DisplaySearchText** and **DisplayKeyboardHint** properties of the **SearchAsType** object are used to hide the corresponding parts of the search window. But if they both are set to False, the search window is not displayed at all — what we need if we want to filter iGrid from code without any additional UI elements.

The **AutoCancel** property specifies whether the current search-as-type operation is automatically cancelled when iGrid loses input focus. This is done by default, which corresponds to the default value of True of this property. However, when filtering is performed from code, the filter should remain visible regardless of input focus, so set this property to False.

The **FilterKeepCurRow** property is used to keep the current row selection if possible after changing visibilities of rows after applying a new filter.

Having all this, now it is easy to filter our grid by a substring from code. To do this, we assign the substring to filter by to the **SearchText** property of the **SearchAsType** object, for example:

```
iGrid1.SearchAsType.SearchText = "abc";
```

This approach can be used to filter iGrid from an external control, such as a **TextBox** control. Its **TextChanged** event is used for that:

```
private void TextBoxSearch_TextChanged(object sender, System.EventArgs e)
{
  iGrid1.SearchAsType.SearchText = TextBoxSearch.Text;
}
```

Pay attention to the fact that iGrid can be filtered by invisible columns this way too. An invisible column can contain some special values to filter by the user should not see, and you can specify this column in the **SearchCol** property of the **SearchAsType** object for filtering.

The **SearchAsType** object also provides methods and properties for controlling and checking search-as-type from code:

- **Cancel** cancels the current search-as-type operation and clears the search string. It is mainly used when the **AutoCancel** property is False.
- **GoNext** and **GoPrev** move the current cell to the next or previous matching cell.
- **HasMatches** indicates whether the current search text has any matches in the grid.
- **IsActive** indicates whether search-as-type is currently active.

22. PROPERTY RESET INFRASTRUCTURE

22.1. Universal Property Reset Functionality

iGrid provides you with universal tools to reset properties to default values at run time. This can be done with the help of the static methods of the **iGPropertyManager** class implemented in the iGrid.NET assembly (TenTec.Windows.iGridLib.<version>.dll). Its methods can be used to reset properties of the core **iGrid** component and other main components, such as **iGDropDownList**, **iGCellStyle**, **iGColHdrStyle**, **iGAutoFilterManager** and **iGPrintManager**.

One good example of when these tools can be useful is given below. Many applications provide the user with the ability to change the look and behavior of some parts of iGrid and revert them to their original state. For example, you may provide your users with the option to adjust the width and color of the grid lines in iGrid and revert them back to the default values. If you have a check box for such a setting, the corresponding event handler may look like the following code snippet:

```
private void checkBoxThickGridLines_CheckedChanged(
  object sender, EventArgs e)
{
  if (checkBoxThickGridLines.Checked)
  {
    fGrid.GridLines.Horizontal.Width = 2;
    fGrid.GridLines.Horizontal.Color = Color.Red;
    fGrid.GridLines.HorizontalLastRow.Width = 3;
    fGrid.GridLines.HorizontalLastRow.Color = Color.DarkRed;
    fGrid.GridLines.Vertical.Width = 2;
    fGrid.GridLines.Vertical.Color = Color.Red;
    fGrid.GridLines.VerticalLastCol.Width = 3;
    fGrid.GridLines.VerticalLastCol.Color = Color.DarkRed;
  }
  else
  {
    fGrid.GridLines.Horizontal.Width = _savedHorizontalWidth;
    fGrid.GridLines.Horizontal.Color = _savedHorizontalColor;
    fGrid.GridLines.HorizontalLastRow.Width = _savedHorizontalLastRowWidth;
    fGrid.GridLines.HorizontalLastRow.Color = _savedHorizontalLastRowColor;
    fGrid.GridLines.Vertical.Width = _savedVerticalWidth;
    fGrid.GridLines.Vertical.Color = _savedVerticalColor;
    fGrid.GridLines.VerticalLastCol.Width = _savedVerticalLastColWidth;
    fGrid.GridLines.VerticalLastCol.Color = _savedVerticalLastColColor;
  }
  fGrid.Invalidate();
}
```

Needless to say that you also need to declare all these _saved* variables in your form and initialize them with the default values of the corresponding properties when your form starts. The universal property reset tools allow you to avoid having such variables and to reset any iGrid property to its default state with one call to the **iGPropertyManager**.**ResetProperty** method. Having this method, we can rewrite the else part in our event handler as follows:

```
private void checkBoxThickGridLines_CheckedChanged(
  object sender, EventArgs e)
{
  if (checkBoxThickGridLines.Checked)
  {
    fGrid.GridLines.Horizontal.Width = 2;
    fGrid.GridLines.Horizontal.Color = Color.Red;
    fGrid.GridLines.HorizontalLastRow.Width = 3;
    fGrid.GridLines.HorizontalLastRow.Color = Color.DarkRed;
    // ... (other custom grid line settings)
  }
  else
  {
    iGPropertyManager.ResetProperty(fGrid, "GridLines");
  }
  fGrid.Invalidate();
}
```

The **iGPropertyManager** class implements the following methods:

```
static void ResetProperty(object obj, string property)

static bool CanResetProperty(object obj, string property)

static IEnumerable<string> GetResettableProperties(object obj)

static bool? IsPropertyModified(object obj, string property)
```

The method names are self-explanatory and make it clear that you can not only set an object's property to its default value (the **ResetProperty** method), but also get info about the property (**CanResetProperty**, **IsPropertyModified**). Note that not all iGrid properties, especially some properties inherited from parent classes like **Control**, provide support for resetting to default values. That's why **iGPropertyManager** also implements the **CanResetProperty** and **GetResettableProperties** methods allowing you to find properties allowing reset.

Pay attention to the fact that all methods are static. In other words, you do not need to create an instance of the **iGPropertyManager** class to call them. In the code snippet above we called the **ResetProperty** method of **iGPropertyManager** without any prior declarations.

More details about each method of the **iGPropertyManager** class are in The iGPropertyManager Methods topic.

22.2. The iGPropertyManager Methods

The ResetProperty method

The **ResetProperty** method of the **iGPropertyManager** class is the main method that can be used to reset any property to its default value, for example:

```
// Reset a property of an enumeration type (iGBorderStyle):
iGPropertyManager.ResetProperty(iGrid1, "BorderStyle");

// Reset a complex object property of the iGGridLines type:
iGPropertyManager.ResetProperty(iGrid1, "GridLines");

// Reset a property storing an array of iGCellStyle items:
iGPropertyManager.ResetProperty(iGrid1, "GroupRowLevelStyles");
```

This method accepts an object whose property is to be reset, along with the property's name specified as a string. The property name is case-sensitive. If the method can't find the specified property, it throws an **ArgumentException** with the message "Property <property> not found".

If you want to provide compile-time check for property names specified in **ResetProperty** method calls, use the **nameof** operator in C# or **NameOf** in VB. Below is a better version of the previous code snippet with compile-time check of property names:

```
iGPropertyManager.ResetProperty(iGrid1, nameof(iGrid1.BorderStyle));
iGPropertyManager.ResetProperty(iGrid1, nameof(iGrid1.GridLines));
iGPropertyManager.ResetProperty(iGrid1,
nameof(iGrid1.GroupRowLevelStyles));
```

The **ResetProperty** method can be called for complex object properties like the **GridLines** property of iGrid. In this case it resets the sub-properties of such properties. You can also call **ResetProperty** for individual sub-properties of a complex property. For example, the following call will reset the **Vertical** property of iGrid1's **GridLines** property to its default value:

```
iGPropertyManager.ResetProperty(iGrid1.GridLines, "Vertical");
```

The equivalent call with compile time check of the property name will be

```
iGPropertyManager.ResetProperty(
  iGrid1.GridLines, nameof(iGrid1.GridLines.Vertical));
```

The CanResetProperty method

The **CanResetProperty** method can be called to know whether a property can be reset. For example, the **IsDocInPrintableArea** property of **iGPrintManager** is a read-only Boolean property indicating whether the document margins are in the area where the selected printer can print. Thus, the following expression will return False:

```
iGPropertyManager.CanResetProperty(iGPrintManager1, "IsDocInPrintableArea")
```

Note that many object properties like **iGrid**.**GridLines** are read-only itself (you cannot replace the whole object property value in the **iGrid**.**GridLines** property), but the sub-properties of these properties can be reset. The **CanResetProperty** method returns True for such object properties indicating that their sub-properties can be reset.

The GetResettableProperties method

The **GetResettableProperties** method returns all properties of a component or object that can be reset. These properties are returned as an enumerable string collection that can be used in a for-each loop or with LINQ extension methods. This allows you to build simple loops to reset a whole component to its default state. Below is an example of how you can set an iGrid instance to the default state as if it had just been created:

```
foreach (var prop in iGPropertyManager.GetResettableProperties(iGrid1))
{
  iGPropertyManager.ResetProperty(iGrid1, prop);
}
```

Pay attention to the fact that iGrid inherits many properties from the **Control** class defined in the **System.Windows.Forms** namespace, such as **Location** and **Size**. The loop above will reset these properties too because these properties are considered iGrid properties and can be reset. If you want to keep the position of iGrid on a form, you can use LINQ's **Except()** to filter the property list like in the following code:

```
string[] layoutProps = new string[] {
  "Anchor", "Dock", "Location", "Size", "Margin" };

foreach (var prop in
  iGPropertyManager.GetResettableProperties(iGrid1).Except(layoutProps))
{
  iGPropertyManager.ResetProperty(iGrid1, prop);
}
```

The IsPropertyModified method

You can determine whether a resettable property has a non-default value with the help of the **IsPropertyModified** method of the **iGPropertyManager** class. This method returns a Nullable Boolean value indicating whether a property was modified (True) or not (False). If the specified property does not provide the corresponding infrastructure tools to know its modification status, the method returns null (Nothing in VB). Note that the ability to know property modification status is based on the same internal tools as the ability to reset a property, and thus it can be also checked with the **iGPropertyManager**.**CanResetProperty** method.

23. RESIZEABLE UI, HIGH DPI AND TABLETS

23.1. Concept of Overridable Properties

iGrid allows you to resize all its elements, such as cell combo buttons and scroll bars, for comfortable use in various scenarios. For example, iGrid elements can be increased for high-resolution screens, for easy use with fingers on tablets or for people with visual impairment when they work with an application on displays with the traditional pixel density.

To provide optimal look on desktop displays out-of-the-box, iGrid calculates a lot of size parameters related to the user interface automatically. Among them are such parameters as the size of the level indent used to show hierarchical row relations and the thickness of the scroll bars. iGrid uses the corresponding system metrics in these calculations, so that their actual values are taken into account if they were adjusted by the user. If the application is DPI-aware, the DPI of the display is also included in these calculations to provide usable iGrid elements on high-resolution displays.

These size parameters set by default can be adjusted for special cases. To do it, iGrid provides you with a set of properties based on a concept of overridable values. The main rules of this concept are as follows:

- You can read the actual value of a UI parameter with the corresponding read-only property. Let's call it simply **Parameter**.
- To use a custom value for this UI parameter instead of the value calculated by iGrid automatically, you assign this value to the property with the "Override" suffix — **ParameterOverride**.
- To retrieve the value calculated by iGrid automatically regardless of the fact whether you overwritten it with the **ParameterOverride** property, read the value of the corresponding read-only **ParameterAutoValue** property.

Let us explain this for the level indent area. iGrid calculates the width of the level indent automatically to provide the best look on the screen taking into account several other parameters, such as the size of the plus/minus button and the current DPI of the screen. The **iGrid**.**LevelIndent** property is a read-only property and it returns the effective width of the level indent calculated automatically by default. If you want to redefine this default behavior and use a specific constant level indent size, you can do it by assigning this size value to the **iGrid**.**LevelIndentOverride** property. The third level indent related property, **iGrid**.**LevelIndentAutoValue**, allows you to retrieve the level indent size iGrid would use automatically even if you overwrote it with **iGrid**.**LevelIndentOverride**.

The key point in this property system is that the **ParameterOverride** property has the **Nullable`1** data type for the underlying value type of the parameter. The default value of any **ParameterOverride** property is null (Nothing in VB), which means that the value calculated automatically is used. When you assign a concrete value to the **ParameterOverride** property, you "ask" iGrid to use namely this value — be it a positive value or even zero. To return to automatic calculation of the value, assign null to **ParameterOverride** again.

For example, to use the level indent of 15 pixels, use the following statement:

```
iGrid1.LevelIndentOverride = 15;
```

To return to automatic calculation of the size of the level indent, do the following:

```
iGrid1.LevelIndentOverride = null;
```

Pay attention to the following screenshots demonstrating how the value of any **ParameterOverride** property looks in the Visual Studio Property Browser when the property contains the default value of null or a given value:

LEVELINDENTOVERRIDE = NULL		LEVELINDENTOVERRIDE = 0	
HotTracking	True	HotTracking	True
ImageList	(none)	ImageList	(none)
LevelIndentOverride		LevelIndentOverride	**0**
MarginAfterLastCol	0	MarginAfterLastCol	0
MarginAfterLastRow	0	MarginAfterLastRow	0

To set the value of such a property to null in the Windows Forms designer, simply erase the property value in the Property Browser and commit this empty value.

23.2. Sizes of Clickable Elements

Default sizes of clickable elements

iGrid provides you with many clickable elements. These are scroll bars with their constituent parts (buttons and thumb); combo buttons, ellipsis buttons and check boxes in cells; plus/minus buttons in group rows and trees; x-buttons in column headers. By default the sizes of all these clickable elements of iGrid are related to the thickness of the scroll bar in the OS. This makes sense because in the general case the system thickness of the scroll bar is set to a value that is comfortable for use on a particular screen, and thus iGrid automatically provides its clickable elements of comfortable sizes too.

The user may explicitly or implicitly change this system setting in the OS to get usable scroll bars in all applications — for example, by setting the display scale factor for a tablet screen to 200% or 250% in the Windows 10 settings. As a result, iGrid also changes the size of its scroll bars, combo and ellipsis buttons accordingly to provide consistent look.

Most iGrid clickable elements are shaped like a square. The side length of a square for a clickable element of a particular type equals the system thickness of the scroll bars or is related to it by some coefficient. The side length of combo and ellipsis buttons is exactly equal to the system scroll bar thickness. The side lengths of cell check boxes and plus/minus buttons are also related to this basic scroll bar thickness, but with factors of 0.8125 and 0.6 respectively. If the base scroll bar size is changed, all cell controls are changed in size harmoniously to provide the best look. The only exclusion is the cell check box when iGrid is rendered with the OS visual styles: this styled UI element never changes its size because the system provides the styled check box bitmap of a fixed size corresponding to the current screen DPI.

The drawing of glyphs inside cell controls (the tick mar in check box, the plus and minus sign in plus/minus buttons, etc.) is implemented in such a way that they are scaled depending on the size of controls. If the size of a control becomes bigger, the glyph pen becomes thicker to provide a better look. The drawing algorithms take traditional 96dpi screens and high-resolution screens to provide perceptible cell control glyphs.

Adjusting sizes of clickable elements

iGrid provides the ability to specify your own base size for all elementary controls in cells and column headers instead of the system scroll bar size used by default as described above. This feature allows you to adjust the size of cell and column header controls for your particular needs — for example, to make them easily usable with fingers on touch screens.

This functionality is provided by the **ElemControlBaseSizeOverride** property of iGrid, which works according to the Concept of Overridable Properties. This is a nullable integer property that specifies the desired base size of elementary controls in pixels. The default value of the property is null, which means that the system scroll bar size is used as the base value to calculate sizes of elementary controls as described above. To specify your own base size, assign the corresponding non-negative value to this property.

The default base size value is returned by the read-only **ElemControlBaseSizeAutoValue** property of iGrid. In the current version it returns the system scroll bar size, but if the base size value provider will be changed in a future version, you can always retrieve it with this property without changing your code. The effective base size value for elementary controls can be retrieved with the read-only **ElemControlBaseSize** property of iGrid.

Pay attention to the fact that you can assign 0 to the **ElemControlBaseSizeOverride** property. If you do so, you will remove all elementary controls from cells. This feature can be used to hide all cell controls with one statement.

Changing thickness of iGrid scroll bars

The thickness of iGrid scroll bars can be set independently from the OS settings. This allows you to make iGrid scroll bars thicker for touch-oriented interfaces or thinner if you want to provide more space for cells.

You can specify your custom height for the horizontal scroll bar and custom width for the vertical scroll bar with the **HeightOverride** and **WidthOverride** properties of the **iGrid**.**HScrollBar** and **iGrid**.**VScrollBar** objects respectively. The default value for these properties is null, which means that iGrid sets the thickness of each scroll bar automatically depending on the system settings. To specify a custom scroll bar thickness in pixels, assign a positive integer value to the corresponding property. For example, the following statement will set the width of the vertical scroll bar to 25 pixels:

```
iGrid1.VScrollBar.WidthOverride = 25;
```

Pay attention to the fact that these two properties allow you to even set the thickness of the vertical scroll bar and horizontal scroll bar independently from each other if required.

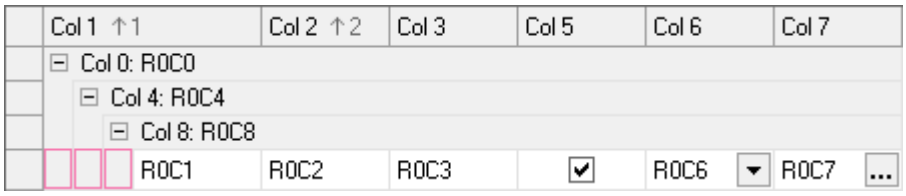
23.3. Sizes of Informational Elements

The size of the sort arrows indicating column sort orders in their headers is related to the size of the header font, which is specified with the **iGrid**.**Header**.**Font** property or inherits **iGrid**.**Font** if not specified. Generally the size of a font is specified in points, which yields to automatic scaling of characters to provide the same linear sizes on screens with different DPI values. As a result, the iGrid sort arrows are changed in size together with the header font and they have the same size comfort for eyes.

The row header glyphs (the triangle of the current row, the pencil icon reflecting editing state, etc.) are automatically scaled according to the current DPI so that the user can recognize them easily.

23.4. Controlling Level Indent Size

The level indent area in iGrid is used to indicate row hierarchy levels. The pink rectangles on the picture below mark every level indent in the 4th row of the grid:



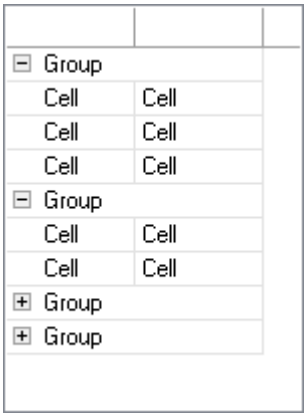
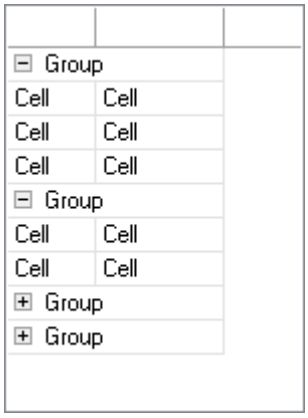
The size of level indent should correspond to the size of the plus/minus button whose size can change depending on many factors. For example, you can change the iGrid appearance with its **RenderStyle** property, the indent around plus/minus button can be adjusted with the **TreeButtonIndent** property of iGrid, custom-drawn plus/minus buttons can be used. iGrid also supports high-resolution screens and the ability to change the size of controls with its **ElemControlBaseSizeOverride** property (see the Sizes of Clickable Elements topic for more

information). All this adds more scenarios in which the size of plus/minus buttons may change dynamically. To free the developer from coding a complex algorithm that calculates the appropriate level indent size, iGrid automatically sets it based on the effective plus/minus button size and indent around it when anything related to the size of these elements has changed.

If needed, this default behavior can be turned off to specify any custom fixed size of level indent. This is done with the help of the **iGrid**.**LevelIndentOverride** property created according to the Concept of Overridable Properties in iGrid. If this property equals null (the default value), iGrid changes the level indent size automatically as described above. To specify your custom level indent, assign the corresponding non-negative value in pixels to this property.

If you need to know the effective size of level indent, you can retrieve it with the **LevelIndent** property of iGrid. The third iGrid property related to level indents and introducing accoring to the concept of overridable properties is **LevelIndentAutoValue**. It returns the optimal size of level indent calculated by iGrid automatically for the best look. You can use this value at any time regardless of the value in the **LevelIndentOverride** property if you need to know the optimal size of the level indent area.

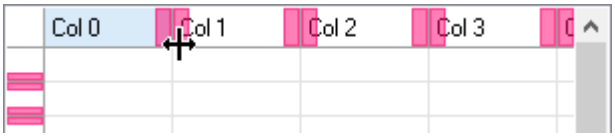
Pay attention to the fact that the **LevelIndentOverride** property accepts non-negative integer values, which includes the zero value. This means that you can completely remove visual level indents in rows on the screen after you set the level indent size to 0. This can be helpful in situations if you use one level of parent-children relations and you want to save some horizontal space by removing level indents in child rows. Compare how a grid with manual group rows can look when the default and zero level indents are used:

DEFAULT LEVEL INDENT:	**ABSENT LEVEL INDENT:**
	

23.5. Column and Row Resize Areas

iGrid provides two settings related to interactive column and row resizing. They allow you to adjust iGrid for more comfortable use on high-resolution screens and touch panels.

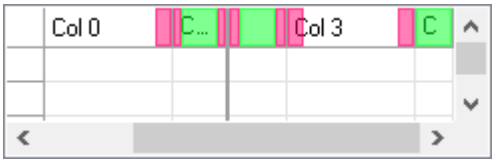
The first of these customizable parameters is the size of the area around column and row dividers in which the mouse pointer turns into the resize cursor to indicate that the user can resize the corresponding object. These resize areas are marked with pink rectangles on the picture below:



The sizes of these areas are configured independently for columns and rows with the integer properties of iGrid named **ColResizeAreaExtentToDivider** and **RowResizeAreaExtentToDivider**. They specify the size of the resize area in pixels on both sides of the column and row dividers and default to 8 and 4 respectively.

The second customizable parameter related to object resizing is the so called "protected space" — the area between two consequent dividers that should not be covered by the resize areas as much as possible. This is an important concept of the object resizing functionality because the user always has some space in a resizable object they can click or grab to drag the object to another place.

If the column width becomes small enough, iGrid automatically decreases the resize areas to provide maximum room for the protected space. The following picture demonstrates this concept:



The same grid with 5 columns from the previous screenshot was modified to demonstrate the concept of protected space. We froze two first columns, decreased the size of the second and fifth columns, and scrolled the grid in the horizontal direction. The pink rectangles mark resize areas, the green rectangles mark protected spaces.

The second column is small enough to provide full resize areas together with the protected space, so the resize areas in this column were decreased. The width of the third column was not decreased, but this column is partially hidden by the frozen area, so its resize areas were also decreased. The last column is so small that it does not provide resize areas at all.

The sizes of protected spaces for columns and rows are set with the integer **ColResizeAreaProtectedSpace** and **RowResizeAreaProtectedSpace** properties of iGrid. They specify the size of protected spaces in pixels and default to 12 and 8 respectively.

23.6. Tablets and Touch Screens Support

Support for tablets and touchscreens is to make it easier to control the application with your fingers. This is mainly done by implementing the following features:

1. Finger gestures that facilitate basic actions.
2. Increasing the size of clickable controls to make them finger-friendly.

Regarding point 1, iGrid provides all main Windows touch screen gestures out-of-the-box. These are:

- Single tap. It is equivalent to a mouse click that selects a cell or performs another action depending on the object under the touch point (clicks a column header to sort it, clicks a combo button in a cell, etc.)
- Double tap. It is equivalent to a double-click that generally puts the clicked cell into edit mode.
- Tap and hold. It works like a right-click and can open an attached context menu or perform another action coded by the developer.
- Tap and drag. You can use this gesture like dragging with the mouse to resize and reorder columns and the like.
- Touchscreen scroll. You can use one finger to drag the contents of iGrid in the direction you want to scroll in.

Regarding point 2, there are many strategies on how to accomplish this. We will demonstrate one of the strategies and write appropriate code that can be used as a base for your real-world application. The implementation will be based on the properties and features described in this section of the manual because they are related directly to the task we are implementing.

Let's write a method we can use to apply touch-friendly settings to any iGrid passed to the method as the parameter. Its header could look like this:

```
private void AdjustGridForTouchScreen(iGrid grid)
{ }
```

The first thing we will need in our method is the current DPI of the screen. The fact is that displays of modern tablets and touch screens can be high-resolution screens, and we must take into account this fact too in our calculations. We can retrieve the needed DPI info like this:

```
float myDpi;
using (Graphics graphics = this.CreateGraphics())
  myDpi = graphics.DpiY;
float myDpiScaleFactor = myDPI / 96;
```

Now let's define a variable that specifies the desired size of a clickable element in inches to make it finger-friendly on the screen. Let it be one-third of an inch:

```
float myFingerFriendlySize = 0.33333f;
```

To make all clickable elements of cells and column headers (combo buttons, x-buttons, etc.) this size, we use the **ElemControlBaseSizeOverride** property of iGrid:

```
grid.ElemControlBaseSizeOverride = (int)(myFingerFriendlySize * myDpi);
```

The thickness of iGrid scroll bars is inherited from the OS by default, so that we need to override it with the same base size value for cell controls:

```
grid.VScrollBar.WidthOverride = grid.ElemControlBaseSizeOverride;
grid.HScrollBar.HeightOverride = grid.ElemControlBaseSizeOverride;
```

To easily resize columns and rows with fingers by dragging column and row dividers, we can adjust the default parameters of the resize areas for the current DPI:

```
grid.ColResizeAreaExtentToDivider =
  (int)(grid.ColResizeAreaExtentToDivider * myDpiScaleFactor);
grid.ColResizeAreaProtectedSpace =
  (int)(grid.ColResizeAreaProtectedSpace * myDpiScaleFactor);

grid.RowResizeAreaExtentToDivider =
  (int)(grid.RowResizeAreaExtentToDivider * myDpiScaleFactor);
grid.RowResizeAreaProtectedSpace =
  (int)(grid.RowResizeAreaProtectedSpace * myDpiScaleFactor);
```

You can also adapt the default width of the row header area to the current DPI:

```
grid.RowHeader.Width = (int)(grid.RowHeader.Width * myDpiScaleFactor);
```

Actually this is all what we need to provide basic support for touch screens. Other settings below can help to enhance the overall picture.

If we do not change the grid font size, cell and column header texts will have the same linear size on any screen. This happens because by default iGrid's font size is set in points, which are device-independent measurement units — 1 inch equals 72 points. However, texts rendered with the default font size can look small compared to cell controls with increased sizes, so let's make the grid text 80% of the size of our clickable controls:

```
grid.Font = new Font(grid.Font.FontFamily,
  myFingerFriendlySize * 0.8f * 72);
```

If you did not change the default font for the header area, it will inherit this font too. And the size of the sort arrows inside column headers will be automatically changed to correspond to the size of column title texts (see the Sizes of Informational Elements topic for more information).

To create a more appealing picture, we can add more space around plus/minus buttons and column headers placed into the group box:

```
grid.TreeButtonIndent =
  (int)(grid.TreeButtonIndent * myDpiScaleFactor);
grid.GroupBox.ColHdrSpace =
  (int)(grid.GroupBox.ColHdrSpace * myDpiScaleFactor);
```

If you plan to call our AdjustGridForTouchScreen() method before adding rows to iGrid, we recommend that you set the default row height to the optimal value according to our above settings by executing the following statement:

```
grid.DefaultRow.Height = grid.GetPreferredRowHeight(true, false);
```

In the case our method is called after you added data to iGrid, we recommend that you automatically fit the size of columns and rows by executing the following code:

```
grid.BeginUpdate();
grid.Cols.AutoWidth();
grid.Rows.AutoHeight();
grid.EndUpdate();
```

The height of column headers is adjusted automatically after changing the header font (see the Setting Header Row Heights topic), so no need to do something for this explicitly.

24. GENERAL PRINCIPLES OF EDITING

24.1. Built-in Cell Editors

iGrid provides you with the built-in editing functionality for its text box, combo box and check box cells. Note that the type of a cell is specified with its **Type** property that accepts only two values — **Text** (text cell) and **Check** (check box cell). Actually a combo box cell is a text cell with a drop-down list attached to it via the **DropDownControl** property, but we consider it as a separate cell type in the context of editing because it provides you with some additional specific features related to drop-down lists.

For all iGrid cell types, the editing is based on the following general schema: you put a cell into edit mode (start editing), then you edit it and after that save your changes (commit editing) or reject them (cancel editing). Depending on the cell type, some parts of this process can be skipped. For example, for a check box cell there is no period of time while you are editing it like a text box cell by typing in text — you simply change the status of the check box control.

The user can start editing with the mouse or keyboard. All the possible actions to do that are listed in the Keyboard and Mouse Editing Commands topic.

24.2. Keyboard and Mouse Editing Commands

General commands to start and finish editing

You can start and finish editing a cell using the following common commands.

To start editing:

- Press ENTER or F2.
- Press F4 to start editing with a drop-down list (only for combo box cells).
- Press a character (a letter, digit, symbol, punctuation, or the space key).
- If single-click edit mode is off, click the current cell or double-click any cell; if on, simply click a cell to edit it (see also Single-Click Edit Mode).

To finish editing and commit changes in the cell:

- Press ENTER or SHIFT+ENTER.
- Press the TAB key to move input focus to the next cell.
- Press SHIFT+TAB to move input focus to the previous cell.
- Click another cell, control on the form, another form, or another application to move input focus to it.

To finish editing and cancel changes in the cell:

- Press the ESC key.
- Scroll the grid or resize it.

Specific keyboard commands for cells of particular types

To starting editing:

ACTION	TEXT CELL	COMBO CELL	CHECK CELL
Pressing the ENTER or F2 key.	The text of the cell is selected and the caret is moved to the end of the cell text.	If the **NoTextEdit** type flag is specified for the cell, the drop-down control is shown. Otherwise the cell text is selected and the caret is placed at the end of the cell text.	The state of the check box control is toggled.
Entering a character (including punctuation marks and the space character).	The entered character replaces the cell text.	If the **NoTextEdit** type flag is specified for the cell, the drop-down control is shown. Otherwise the entered character replaces the cell text.	The state of the check box control is toggled.
Pressing the F4 key.		The drop-down control is shown.	

To commit editing:

ACTION	TEXT CELL	COMBO CELL
Pressing the ENTER or SHIFT+ENTER key.	If the cell allows entering multiline text, the ENTER key is used to create a new text line and you can commit editing only if you press the SHIFT+ENTER key combination.	If the cell is being edited with a drop-down control, the drop-down control is closed and the selected value is saved to the cell. Otherwise, if the cell allows entering multiline text, the ENTER key is used to create a new text line, and you can commit editing only if you press the SHIFT+ENTER key combination.

Notes

When iGrid is in multi-selection mode, some of the actions listed above change selection instead of starting editing. If multi-simple selection mode is on, editing cannot be started by clicking a cell or by pressing the space character because these actions change selection. In multi-extended selection mode, if the user clicks a cell to start editing, all the modifier keys (CONTROL, SHIFT, ALT) should be released and the cell under the mouse pointer should be the only selected cell or should not be selected before clicking — otherwise the cell or a group of cells change their selection state. Similarly, if the user presses the SPACE key in multi-extended selection mode, editing is started only if all the modifier keys are released.

If the user clicks a scroll bar button while a cell is being edited, the edited cell loses input focus. As a result, the grid tries to commit the changes. If the changes have not been committed and the grid is scrolled, the changes will be canceled and editing finished.

24.3. Single-Click Edit Mode

When we consider editing with the mouse, iGrid offers 2 options on how to start editing a cell. When you use iGrid with the default settings, the user should first click the cell to make it current and then click it again to start editing. An alternative way to do this is to double-click the cell (the first click makes the cell current, the second click puts the cell into edit mode). You can simplify editing for the user so that they can start editing with a single click on the desired cell without first selecting it. This mode is called single-click edit mode.

Single-click edit mode can be enabled for the entire grid or individual cells. To enable it for the entire grid, set the Boolean **SingleClickEdit** property of iGrid to True. To enable single-click edit mode for individual cells, use their **SingleClickEdit** property. The **iGCell**.**SingleClickEdit** property has the **iGBool** type that allows you to inherit the single-click edit mode setting from the parent object, i.e. the column cell style or the whole grid.

The ability to enable single-click edit mode for individual cells can be very useful for fine-tuning editing in the grid — for example, if your grid contains check box cells. Let's suppose the second column in your grid contains check box cells to enable/disable different options. If iGrid is used with the default settings, a click on a check box cell just selects the cell without changing the check box state — which can be useful in many cases (for example, to show detailed information about the selected option and the like). However, toggling an option in this case will require 2 clicks. The first click will select the corresponding check box cell, and only the second click on the same cell will toggle the check box state. The user may find this approach tedious, especially if they need to switch many options. The best solution in this case is to enable single-click edit mode for the cells in the column with options so that any check box will toggle its state right after the user clicked the cell:

```
iGrid1.Cols[1].CellStyle.SingleClickEdit = iGBool.True;
```

In Windows, if the user opens the drop-down list of a combo box control, a click outside of the drop-down list closes the list without any further action associated with the mouse click. By default, iGrid behaves the same way: if the user clicks outside of a drop-down control opened from a combo box cell, this action is treated as a command to close the drop-down control and do nothing more. This can lead to user misunderstanding if single-click edit mode is on: if the user opened a drop-down control in a cell and clicks another cell to activate editing in it, only the drop-down control will be closed and the user will need to click the other cell again to activate editing.

The Boolean **SingleClickEditOnDropDownClose** property of iGrid can help to solve this problem. Its value specifies whether iGrid will activate editing in the clicked cell if the user clicks outside of a drop-down control when single-click edit mode is enabled for the clicked cell. The default value of this property is False that corresponds to the iGrid behavior described above. To enable editing with a single click when this click closes an opened drop-down control, set this property to True.

24.4. Common Events Related to Editing

When the user initiates a cell edit, iGrid first raises the **RequestEdit** event. This event allows you to know the cell the user is going to edit and to adjust the text that will be edited if required. You can also prohibit edit in the cell by assigning False to the **DoDefault** property of the event arguments.

As it was described in the Keyboard and Mouse Editing Commands topic, the user can perform different actions to finish cell edit and commit cell changes. When one of these actions is performed, iGrid determines whether the changes must be committed according (see the Commit Edit Conditions topic for more details). If the changes must be committed, iGrid raises the **BeforeCommitEdit** event before doing this. This event allows you to check the entered value and perform one of the following actions by setting the **Result** property of the event arguments to one of the following values:

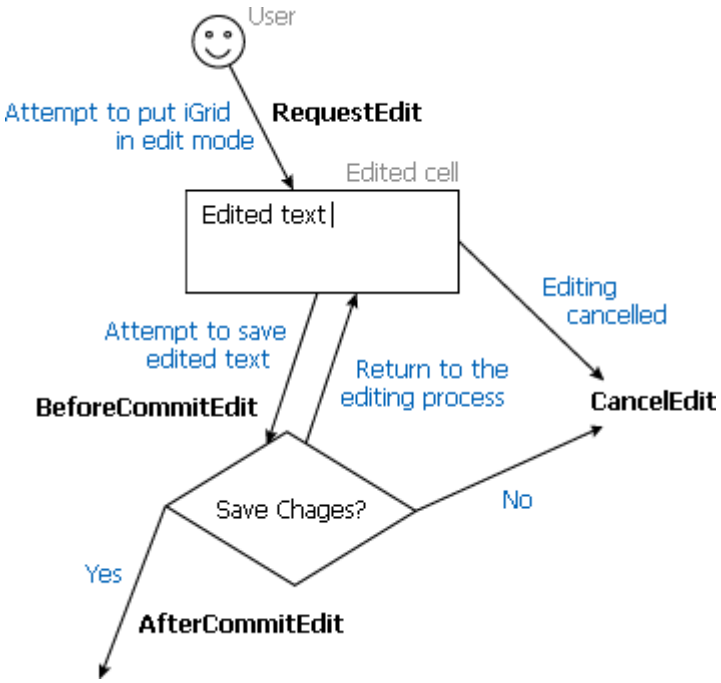
- **Commit** — iGrid will finish edit and save the new value.
- **Cancel** — iGrid will finish edit without saving the new value.
- **Proceed** — iGrid will continue edit.

Note that the value stored in the cell can be of any .NET data type (integer, date, etc.) and it may differ from the entered string in the general case. The value that will be saved in the cell is passed in the **NewValue** property of the event arguments of the **BeforeCommitEdit** event. This value is determined by iGrid based on a set of rules described in greater detail in the Getting Cell Values

from Entered Strings topic. You can also provide your own new cell value in the **NewValue** property in an event handler of the **BeforeCommitEdit** event.

After editing has been finished, iGrid raises the **AfterCommitEdit** event if the changes have been committed or the **CancelEdit** event if the changes have been cancelled.

The following schema demonstrates the relationships between these events:



Pay attention to the fact that this is the universal editing event model applicable to any type of cell. For example, when the user clicks the combo button in a combo box cell to edit it by selecting an item from the drop-down list, the **RequestEdit** event is raised allowing you to disable this kind of editing. The same is true for a check box cell: **RequestEdit** is raised before changing the check box cell when the user clicks it. After the state of the check box cell has been changed, the **AfterCommitEdit** event notifies you about that.

24.5. Commit Edit Conditions

After the user put a cell into edit mode, they have several options to finish editing, which iGrid will interpret as an attempt to save possible changes to the cell. The most used of them are pressing ENTER or clicking outside the cell (see the Keyboard and Mouse Editing Commands topic for more details). However, committing changes are not always necessary in such cases. For example, if the user puts a cell into edit mode just to copy some text from it but not to change it, clicking outside the cell should not always commit edit that did not actually happen. The **CommitEditCondition** property of iGrid specifies the condition under which iGrid commits cell edit in such cases.

This property accepts one of the following values from the **iGCommitEditCondition** enumeration:

VALUE	DESCRIPTION
Always	iGrid always commits edit, even if nothing has changed in the cell.
TextEdited	iGrid commits edit if the user edited the cell text, but it may remain the same when the edit finishes.
TextChanged	iGrid commits edit if the user edited the cell text and it changed.

VALUE	DESCRIPTION
ValueChanged	iGrid commits edit if the user changed the cell text and the value associated with it changed.
Never	iGrid never commits edit even if the user changed the cell.

The default value of the **CommitEditCondition** property is **Always**. It means that iGrid will always try to commit cell edit and raise the **BeforeCommitEdit** event on all appropriate user actions even if nothing has changed in the cell.

The **TextEdited** condition can help to track situations in which the user wants to force some updates even if the same text/value was entered. This condition differs from **Always** because in the case of **Always** the user may not change anything in the cell to commit edit. In the case of **TextEdited** the user must do something to change the cell text (for example, remove the last character and enter it again) to commit edit.

The difference between the **TextChanged** and **ValueChanged** conditions becomes clear if one value may have several text representations. As a rule, this is the case if you format numeric or date-time values with format strings. For example, if a cell contains numeric data, "100.0" and "100" entered into the cell will generate the same value of 100. In the case of **TextChanged** the edit will be committed, in the case of **ValueChanged** — not.

The **Never** condition can be used to implement a kind of read-only mode for the whole grid. If the **CommitEditCondition** property of iGrid is set to **Never**, the user will be able to put iGrid cells into edit mode to copy/paste cell texts or do anything with them, but any user changes will be always ignored when edit finishes.

24.6. Getting Cell Values from Entered Strings

When the user commits changes in a cell allowing text editing, iGrid tries to convert the entered string to the new cell value according to the following rules in the order they are listed below. If a rule has been applied, the process stops.

1. If the user has entered an empty string, the value that will be saved to the cell depends on the **EmptyStringAs** property of the cell. By default an empty string is saved as null (Nothing in VB).
2. iGrid tries to figure out what the cell value type should be. First it checks whether a type is specified in the **ValueType** property of the cell. If it is not specified, iGrid checks whether the cell already has a value, and if so, the type of the existing value will be used. If the cell value type has been determined, iGrid converts the entered string exactly to the value of this type.
3. The last rule means that the cell value type isn't specified and the cell value is currently empty. If the entered string can be converted to a number, an appropriate data type is used to store this number as the new cell value. Otherwise the entered string is saved "as is" as the new string value of the cell.

Some technical notes and tips on the algorithm are outlined below.

Step 2: Specifying a cell value type for the whole column

As a rule, all values in a column have the same data type. You can use the **CellStyle** object of the column to specify the desired type of values in C# the following way:

```
iGrid1.Cols.Add("val", "Value").CellStyle.ValueType = typeof(string);
```

The equivalent code for VB:

```
iGrid1.Cols.Add("val", "Value").CellStyle.ValueType = GetType(String)
```

This will ensure you that even if the user has entered something looking as a number, Step 3 of the algorithm above will not be applied and the entered string will be saved "as is" in the cell.

Step 2: Converting entered string to the desired data type

iGrid tries to convert the entered string to the required type using **Parse** methods of the built-in .NET data types (such as **Int32**.**Parse**). If the conversion cannot be done, an exception occurs. In this case iGrid catches the exception and displays its built-in message box with the exception message and the cell remains in edit mode.

This default behavior can be changed with the **SilentValidation** property. The default value of this property is False and the standard iGrid error message box is displayed. If you set this property to True, you suppress that error message and have the ability to display your own error message and/or convert the entered string to the new cell value according to your custom conversion algorithm. You can do it in the **BeforeCommitEdit** event. If **SilentValidation** is True, iGrid passes the conversion exception in the **Exception** filed of this event's arguments and do not display its built-in error message box. To know whether the entered value cannot be converted to the required data type, you check the **Exception** filed in the event handler of the **BeforeCommitEdit** event, and if it is not null, display your own error message and/or process the entered string using your own algorithm. The entered string is stored in the **NewText** field. The new value you want to store in the cell can be specified in the **NewValue** field.

Below is an example of the **BeforeCommitEdit** event handler you can use as a basis if you need to display your own error message:

```
private void fGrid_BeforeCommitEdit(
  object sender, iGBeforeCommitEditEventArgs e)
{
  if (e.Exception != null)
  {
    Label1.Text = e.Exception.Message;
    e.Result = iGEditResult.Proceed;
  }
}
```

In the code above, we display the conversion error message in a label instead of the standard iGrid message box and tell iGrid that we continue the editing process by setting the **Result** field to **Proceed**.

Step 3: Determining an appropriate data type for a numeric value

Trying to figure out whether the entered string can be saved as a number, iGrid calls the **TryParse** methods of the following .NET data types in the order they are listed: **Int32**, **Int64**, **Decimal**, **Double**. If a call to **TryParse** for a particular type returns True, iGrid uses that data type to store the numeric value.

iGrid calls the overloaded versions of the **TryParse** methods that accept the **NumberStyles** parameter. iGrid passes **NumberStyles**.**Any** as the value of this parameter to allow entering of numeric values using rich formatting rules of the current culture (the currency symbol, digit grouping symbol, etc.) This helps user to enter values or even copy formatted values from other applications. For example, if iGrid works in the en-US culture, the string "$1,234,567.89" will be automatically recognized as a number and will be saved as a Decimal value of 1234567.89.

24.7. Protecting Cells from Editing

To allow or prohibit editing for all cells in the grid, use the **ReadOnly** property of iGrid. To allow or prohibit editing for the cells in individual columns or independent block of cells, use the **ReadOnly** property of the corresponding **iGCellStyle** object or a cell.

For example, if you want to prohibit editing of an individual cell, do the following:

```
iGrid1.Cells[rowIndex, colIndex].ReadOnly = iGBool.True;
```

You can also use cell style objects to disable editing in several cells:

```
iGCellStyle myStyle = new iGCellStyle;
myStyle.ReadOnly = iGBool.True;
iGrid1.Cells[rowIndex1, colIndex1].Style = myStyle;
iGrid1.Cells[rowIndex2, colIndex2].Style = myStyle;
iGrid1.Cells[rowIndex3, colIndex3].Style = myStyle;
```

The following code is typically used to prevent the entire column from editing:

```
iGrid1.Cols[colIndex].CellStyle.ReadOnly = iGBool.True
```

Pay attention to the fact that the **ReadOnly** property has the **iGBool** type that allows you to disable editing, enable editing, or inherit this setting from the parent object due to the special **iGBool**.**NotSet** value.

Editing in individual cells can also be disabled dynamically based on some conditions evaluated at the start of editing. For example, you can analyze the value of another cell and enables editing only if some values are stored in that cell. This functionality is implemented with the **RequestEdit** event and the **DoDefault** field of the event's data object. Below is an example demonstrating how to disable editing in the cells of the first column if the cell from the second column in the same row already has a value:

```
private void iGrid1_RequestEdit(object sender, iGRequestEditEventArgs e)
{
  if (e.ColIndex == 0)
  {
    if (iGrid1.Cells[e.RowIndex, 1].Value != null)
    {
      e.DoDefault = false;
    }
  }
}
```

24.8. User Input Validation in Windows Forms and iGrid

The WinForms .NET infrastructure provides a good and reliable data validation system for controls hosted on a form. It is based on the **Validating** event of the **Form** class. iGrid's editing infrastructure is also built on this .NET functionality.

The default behavior of this validation functionality can be changed with the **AutoValidate** property of the form. This setting affects all controls on the form, including iGrid too. When you change the default value of **EnablePreventFocusChange** to another value like **Disable**, this affects the editing functionality in iGrid and may lead to incorrect behavior. For example, a new value entered into a text cell will not be saved. Another possible drawback is that the general editing-related events like **BeforeCommitEdit** will not be raised.

The same applies to iGrid's **CausesValidation** property, which it inherits from the base **Control** class. The **CausesValidation** property must be set to True (the default value) if text editing is allowed in iGrid. However, if your grid is not editable, you can set this property to False if you want to suppress the standard **Validating** and **Validated** events for the control.

25. BUILT-IN EDITING

25.1. Text Box Cells

25.1.1. Text Box Cell Editing Options

When you edit a text box cell, iGrid can automatically wrap text in it. This setting comes from the text output flags of the cell. By default word wrapping is off, but you can enable it with the **WordWrap** flag specified in the **TextFormatFlags** property of the cell or a style object applied to it.

Another useful option you can enable in text cells is multiline editing. By default, you cannot type in multiline texts into a cell. To enable this ability, you need to specify the **MultilineEdit** flag in the **TypeFlags** property of the cell. In this case the ENTER key will be used to create a new line of text but not to commit editing, and the editing can be committed only by the SHIFT+ENTER combination.

The editing in the cells of this type can also be controlled through some iGrid events whose names start with "TextBox". These are such events as **TextBoxKeyPress** and **TextBoxFilterChar**. Perhaps, the latter one is the most universal and powerful event as it allows you to substitute the entered characters on the fly or even suppress them. This feature can be used to allow the user to enter only a limited number of characters. See the examples of the use of this event in the topic devoted to it.

25.1.2. Text Edit Events and the TextBox Property

iGrid uses the standard .NET **TextBox** control to provide text editing in its text and combo cells. When the user starts to edit text in a cell, iGrid places this text box over the cell, puts the text to edit into it, and activates it (makes it visible and moves input focus to it). After editing is finished, the text box is hidden.

iGrid informs you about various stages of the text edit process and allows you to control keyboard input with its events. The names of these events start with "TextBox".

As a rule, the **TextBoxEditStarted** event is raised first after the text box editor appears on the screen. You can process this event to initialize something related to text editing.

The following events allow you to control keyboard input in the text editor:

EVENT	DESCRIPTION
TextBoxKeyDown	Occurs when a key is pressed while editing a text cell.
TextBoxKeyPress	Occurs when a key is pressed while editing a text cell.
TextBoxKeyUp	Occurs when a key is released while editing a text cell.
TextBoxFilterChar	Occurs before a char key is processed by the text box in a text cell. Allows you to filter input chars.

The **TextBoxKeyDown**, **TextBoxKeyPress**, and **TextBoxKeyUp** events work like the corresponding keyboard processing events in traditional .NET controls. However, there are some points specific for the iGrid infrastructure:

- If iGrid is in browse mode and the user presses a key to start editing, the **TextBoxKeyDown** and **TextBoxKeyPress** events are not generated for this key because it is processed by iGrid itself in its **KeyDown** and **KeyPress** events.
- To ignore a key down event for a key, set the **DoDefault** field of the event arguments object of the **TextBoxKeyDown** event to False (the same for **TextBoxKeyPress**). Note that this

is not the same as suppressing the **TextBoxKeyPress** event, which is done with the help of the **SuppressKeyPress** field of the event arguments object of the **TextBoxKeyDown** event.

- If the user presses a key that finishes editing (ESC or ENTER), only the **TextBoxKeyDown** event is raised for this key because iGrid hides the text editor in its internal KeyDown event handler for the text editor control.

Taking into account these points, the **TextBoxFilterChar** is the most universal and powerful event to adjust text input on the fly. This event allows you to substitute the entered characters or even suppress them, including the first pressed char that starts editing and the char that can't be processed in the **TextBoxKeyDown**/**TextBoxKeyPress** events for the reason described above. Below is an example demonstrating how to allow entering only letters with automatic conversion of allowed characters to upper case:

```
private void iGrid1_TextBoxFilterChar(
  object sender, iGTextBoxFilterCharEventArgs e)
{
  if (char.IsLetter(e.Char))
    e.Char = char.ToUpper(e.Char);
  else
    e.Char = (char)0;
}
```

The **TextBoxTextChanged** event occurs every time when the text in the editor is changed. Note that this event can be raised before **TextBoxEditStarted** if the text box editor is initialized with a text that differs from the current cell text.

The **TextBox** property of iGrid publishes some properties of the text box used to edit text cells. These properties are:

PROPERTY	DESCRIPTION
Text	Gets or sets the text displayed in the text box (which the user has entered to the cell).
SelectionStart	Gets or sets the starting point of text selected in the text box.
SelectionLength	Gets or sets the number of characters selected in the text box.
SelectedText	Gets or sets a value indicating the currently selected text in the text box.

You can access these properties while editing text from such events as **TextBoxFilterChar** or **TextBoxKeyPress** to get the information about the current state of the built-in cell text editor. In the following example, we allow the user to enter up to 5 characters in text cells:

```
private void iGrid1_TextBoxFilterChar(
  object sender, iGTextBoxFilterCharEventArgs e)
{
  if (iGrid1.TextBox.Text.Length == 5)
    e.Char = (char)0;
}
```

iGrid also implements the following events to inform you about mouse actions in the text editor: **TextBoxMouseMove**, **TextBoxMouseDown**, **TextBoxMouseUp**, **TextBoxMouseClick**, **TextBoxMouseDoubleClick**, **TextBoxMouseEnter**, **TextBoxMouseLeave**,**TextBoxMouseHover**.

25.2. Combo Box Cells

25.2.1. Combo Box Cell Basics

A combo box cell is a cell that has a drop-down control containing values the user can select and save in the cell. As a rule, a combo box cell displays the combo button to open the drop-down control for selection.

In most cases the cell drop-down control is a list of items provided by the built-in drop-down list control. But iGrid does not limit you by using only lists in this scenario. For example, you can implement your own drop-down control that will not look like a list of items to select (see Custom Drop-Down Controls). The remainder of this section is dedicated to using the built-in drop-down list control in combo box cells, but keep in mind that all described features can also be used with a custom drop-down control.

The very first step in creating combo box cells is the creation of a drop-down control for these cells. The **iGDropDownList** class is the built-in implementation of the drop-down list control allowing you to edit a cell with a list of items. We will use it in the examples in this section.

Below is a basic sample code showing how to create a drop-down list and populate it with items:

```
iGDropDownList myDropDownList = new iGDropDownList();
myDropDownList.Items.Add("Item1");
myDropDownList.Items.Add("Item2");
myDropDownList.Items.Add("Item3");
```

iGDropDownList objects can be created and populated using code and also at design time. In this case you will not need to write code like the above as it will be created automatically by the Windows Forms Designer. Note that if you create an instance of the **iGDropDownList** class at run time from your code, you should also call the **Dispose** method of this class to correctly clear all used resources and avoid resource leaking when you no longer need this component. As a rule, it is done when closing or unloading the form from memory.

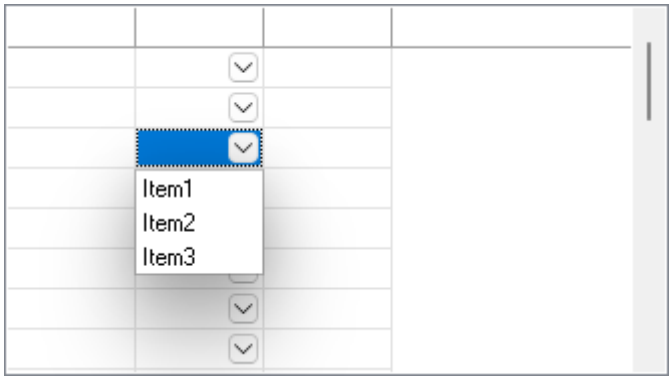
Having a drop-down list, we can attach it to a cell by saving a reference to the drop-down list in the **DropDownControl** property of the cell. For example, we can turn the very first cell into a combo box cell with the following statement:

```
iGrid1.Cells[0, 0].DropDownControl = myDropDownList;
```

As a rule, a drop-down list is used in all cells of a column. The column's cell style object is often used to implement this task in one statement:

```
iGrid1.Cols[1].CellStyle.DropDownControl = myDropDownList;
```

A grid with this drop-down list may look like the following one after opening the list in the third cell of the column with combo box cells:



The full source code of the sample is below:

```
iGDropDownList myDropDownList = new iGDropDownList();
myDropDownList.Items.Add("Item1");
myDropDownList.Items.Add("Item2");
myDropDownList.Items.Add("Item3");

iGrid1.Cols.Count = 3;
iGrid1.Cols[1].CellStyle.DropDownControl = myDropDownList;

iGrid1.DefaultRow.Height = iGrid1.GetPreferredRowHeight(true, false);
iGrid1.Rows.Count = 30;
```

Draw your attention to the **GetPreferredRowHeight** method call. The default row height in iGrid is enough to show one line of text rendered with the default font, but it may be not enough to display full size combo buttons rendered with the OS visual styles (the default rendering style). The **GetPreferredRowHeight** call returns the optimal row height to display both cell text and full size combo buttons.

To open the drop-down list in a combo box cell, click the combo button inside the cell. You can also make the cell current and press F4. Other keyboard commands related to combo box cells can be found in the Keyboard and Mouse Editing Commands topic.

A drop-down list item may have text and image, and both parts are optional. This allows you to create drop-down lists containing only text items, only image items or combine image and text in items. In the general case you can even mix items with text and images and only text items in the same list.

A drop-down list item also has a value that may differ from its text representation on the screen, though in many cases they are the same. When you select a drop-down list item, the value of the selected item is saved in the cell's **Value** property and the text of the selected item is displayed in the cell. If the selected item has an image, it is saved in the **ImageIndex** property of the cell and this image is also displayed in the cell.

In code drop-down list items are represented with instances of the **iGDropDownListItem** class. When you select a drop-down list item in a combo box cell, a reference to the corresponding **iGDropDownListItem** object is saved in the **AuxValue** property of the cell. You can find out more about this process from the Relation Between Combo Box Cell and Its Drop-Down List topic.

25.2.2. General Features of Combo Box Cells

By default, the user can edit the combo box cell as text or select a value from the drop-down list in the cell. If required, you can allow editing the combo box cell only by selecting items from the drop-down list. To do that, specify the **NoTextEdit** flag in the **TypeFlags** property of the cell. The following example shows how to disable text typing in the combo box cells in the first column:

```
iGrid1.Cols[0].CellStyle.TypeFlags = iGCellTypeFlags.NoTextEdit;
```

When you attach a drop-down control to a cell, iGrid automatically displays a combo button inside the cell to open that drop-down. You can hide this combo button by specifying the **HideComboButton** flag in the **TypeFlags** property of the cell. The combo button will be removed from the cell, but the drop-down control can still be opened by pressing the F4 key or an alpha-numeric key (see the Keyboard and Mouse Editing Commands topic).

If the combo button was not removed from the cell with the **HideComboButton** type flag, you can use another grid setting to display this cell element dynamically only in the current cell. This is done with the help of the **ShowControlsInAllCells** property of iGrid:

```
iGrid1.ShowControlsInAllCells = false;
```

Drop-down lists are used to edit cell values by selecting them from a predefined list. Thus, most iGrid methods and events related to editing work for combo box cells too. One such method that work for combo box cells as well is **RequestEditCurCell**. You can use it to display the drop-down list in a combo box cell from code. Editing is always performed for the current cell, and first you should make the target cell the current cell. Then you call the overloaded version of the **RequestEditCurCell** method with the **openDropDown** parameter and specify True as the argument:

```
iGrid1.SetCurCell(2, 1);
iGrid1.RequestEditCurCell(true);
```

Traditional editing events also work for combo box cells. For example, iGrid raises the **RequestEdit** event when the user is trying to activate editing in a cell. This event can be used for combo box cells to get a notification about the moment when the user is about to open a drop-down list. And as in the case of traditional text editing, the **DoDefault** field of this event can be used to prevent the drop-down list from opening after evaluating some conditions.

Another iGrid editing-related event that works for combo box cells is **AfterCommitEdit**. This event can be used to get a notification that the user has selected an item in the drop-down list and thus completed editing in the cell, for example:

```
private void iGrid1_AfterCommitEdit(
  object sender, iGAfterCommitEditEventArgs e)
{
  MessageBox.Show($"Selected value: {e.NewValue}");
}
```

For more information about this event and other editing-related events applicable to combo box cells, see the Common Events Related to Editing topic.

25.2.3. Built-in Drop-Down Lists

The **iGDropDownList** class is intended to create drop-down lists which can be assigned to combo box cells. The table below lists the main properties of the **iGDropDownList** class:

PROPERTY	DESCRIPTION
AutoWidth	Gets or sets a value indicating whether the width of the drop-down list is calculated dynamically to show all the items without clipping.
Font	Gets or sets the font used to display the items in the drop-down list. If not set, the cell font is used.
ImageList	Gets or sets the image list that contains the images displayed in the drop-down list. If an image list is not assigned to a cell, this property will also be used to display the image in the cell.
Items	Gets the collection of the items of the drop-down list.
MaxVisibleRowCount	Gets or sets the maximal number of rows visible in the drop-down list simultaneously.
SearchAsType	Get an object which allows you to set up the search-as-type functionality used in the drop-down list.

The items of a drop-down list are stored in its **Items** property. It is a collection of specific drop-down items that can be populated at design time or run time. An example of population of this collection can be found in the Combo Box Cell Basics topic.

By default, the width of a drop-down list equals the width of the cell from which it is opened. In this case not some drop-down list items may not be fully visible due to insufficient width. You can set the **AutoWidth** property of the drop-down list to True to automatically adjust the list width so that all items will be displayed without clipping.

The height of the drop-down list is determined according to the **MaxVisibleRowCount** property. If the total number of items is greater than the value of this property, the vertical scroll bar will be shown. Otherwise, the height will be equal to the height needed to show all the items entirely.

The font of the items displayed in a drop-down list depends on the font of the edited cell and the **Font** property of the drop-down list. If the **Font** property is not assigned, the drop-down list uses the font of the cell. If the **Font** property is assigned, it is used.

25.2.4. Items in Built-in Drop-Down Lists

Every item of a built-in drop-down list is an instance of the **iGDropDownListItem** class with the following properties:

PROPERTY	DESCRIPTION
ImageIndex	Gets or sets the index of the image displayed in the drop-down list item. The value of this property is saved to the **ImageIndex** property of the cell when the drop-down list item is selected.
Text	Gets or sets the text displayed in the drop-down list item. This property is also used to display the text in the cells which the drop-down list item is attached to.
Value	Gets or sets the value saved to a cell when the item in the drop-down list is selected. If the **Text** property of the item is null (Nothing in VB), this property is used to display the text in the drop-down list.
Visible	Get or sets a value that determines whether the drop-down list item is visible.

Actually every drop-down list item has the invisible part, or value, and the visible part consisting of an image and text. The item image and text are optional — you can use either one of them or both. In many cases strings are used as values and texts like in the basic example from the Combo Box Cell Basics topic. But iGrid.NET allows you to use values of any .NET data type as values of drop-down list items. These can be numbers, items of an enumeration or objects of any class. If the item's value is not a string, iGrid.NET uses the **ToString** method of the value to get its text representation on the screen. You can also define your own string representation of the item with its **Text** property.

The following example demonstrates how to add enumeration members to a drop-down list and specify meaningful texts for them. The special overloaded version of the **Add** method of the **Items** collection in which you can specify value and text is used for that:

```
// the declaration of the enumeration and the drop-down list
enum TableAction {Add, Edit, Delete};

private iGDropDownList myDDL = new iGDropDownList();

private void Form1_Load(object sender, System.EventArgs e)
{
  myDDL.Items.Add(TableAction.Add, "Add a new record");
  myDDL.Items.Add(TableAction.Edit, "Edit the current record");
  myDDL.Items.Add(TableAction.Delete, "Remove the current record");
  iGrid1.Cols[0].CellStyle.DropDownControl = myDDL;
}
```

This approach can be used to create a special text representation for string data. Look at the following definition of a drop-down list:

```
iGDropDownList myCountryList = new iGDropDownList();
myCountryList.Items.Add("CA", "Canada");
myCountryList.Items.Add("RU", "Russia");
myCountryList.Items.Add("UA", "Ukraine");
myCountryList.Items.Add("US", "United States");
...
```

It can be useful if you edit a table that contains a Country field in your grid, and the Country field should accept a two-character key of a country. If you use the list above, you will see full country names such as "Canada" or "Ukraine" on the screen, but the cells which refer this list will store "CA" or "UA" respectively.

The usage of images in built-in drop-down lists is discussed in the topic titled Using Images in Built-in Drop-Down Lists.

25.2.5. Using Images in Built-in Drop-Down Lists

Every item in a built-in drop-down list can display an image. The **ImageIndex** property of an item determines the index of the image within the drop-down list's image list. The reference to this image list is stored in the **ImageList** property of the drop-down list object.

The **ImageIndex** property is also used to display images in the cells which the drop-down list is assigned to. When the user selects an item in a drop-down list, the image index is saved to the cell's **ImageIndex** property.

The image list assigned to a drop-down list can also be used to display the images in the cells which the drop-down list is assigned to. The following ordered list shows the hierarchy of the image lists used to display the image in a cell:

1. The image list of the cell's style
2. The image list of the row's style
3. The image list of the column's style
4. The image list of the cell's drop-down list
5. The image list of the grid

First, iGrid looks for the image list in the cell's style. If it is assigned, it is used. Otherwise, iGrid looks for the image list in the row's style. If it is assigned, it is used, and so on...

If you want to display the same images in a drop-down list and in the cells which the drop-down list is attached to, you should attach the image list to the drop-down list only:

```
...
iGDropDownList1.ImageList = imageList1;
iGrid1.Cols[0].CellStyle.DropDownControl = iGDropDownList1;
...
```

If you want to display images from one image list in a drop-down list and images from another image list in the cells which the drop-down list is attached to, should attach different image lists to the cells and the drop-down list:

```
...
iGDropDownList1.ImageList = imageList1;
iGrid1.Cols[0].CellStyle.DropDownControl = iGDropDownList1;
iGrid1.Cols[0].CellStyle.ImageList = imageList2;
...
```

This capability allows you to implement the following effect. For example, you can display small icons with your items in the drop-down list and the corresponding large images in the cells when an item from the drop-down list is selected.

Note that if you attach an image list to the cells and do not attach an image list to the drop-down list attached to the cells, the drop-down list will not show images.

25.2.6. Methods and Events of Built-in Drop-Down Lists

The FillWithData method

The drop-down list provides you with the only method, **FillWithData**, which allows you to populate the list from one of the following data sources: **IDbCommand**, **IDataReader**, **DataView**, and **DataTable**. The overloaded versions of this method have two or three parameters. The first parameter is always used to specify a data source, and the second parameter always specify the column in the data source which values will be used as the values of the items in the drop-down list. The third optional parameter allows you to specify the column used as text for the drop-down list items.

The following example shows how to fill in a lookup drop-down list from a SQL command; the ID column is used as the item values and the Name column is used as their text representations on the screen:

```
//Create the command
SqlCommand myCommand = new SqlCommand(
  "SELECT id, name FROM Countries", sqlConnection1);

//Create the lookup drop-down list
iGDropDownList myCountryList = new iGDropDownList();
myCountryList.FillWithData(myCommand, "id", "name");
```

The SelectedItemChanged event

The **iGDropDownList** class raises an event called **SelectedItemChanged**. It is raised when the user moves the mouse pointer over the items of a drop-down list, and the selected item under the cursor is changed. This event is useful if you want to display additional descriptions for your drop-don list items (for instance, in a status bar).

Note that this is not the event raised when the user selects an item to put it into the cell and the drop-down list is closed. Use the **AfterCommitEdit** event of iGrid to process this case.

25.2.7. Search-as-Type in Drop-Down Lists and Autocomplete in Cells

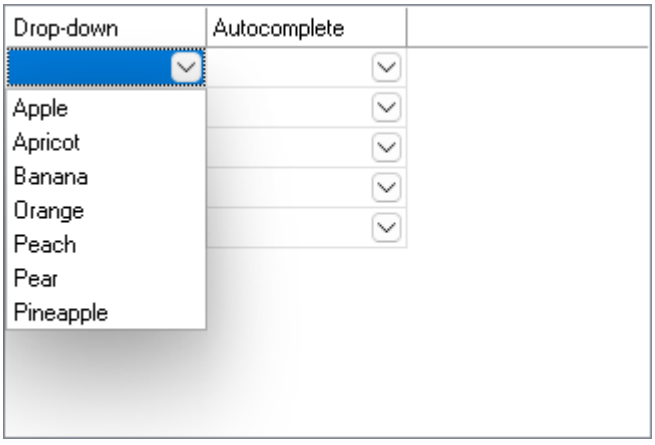
Two modes of built-in drop-down list usage

Chapter Search-as-Type Functionality describes the search-as-type functionality available for grid cells. Built-in drop-down lists provide you with the same functionality helping to find required items faster.

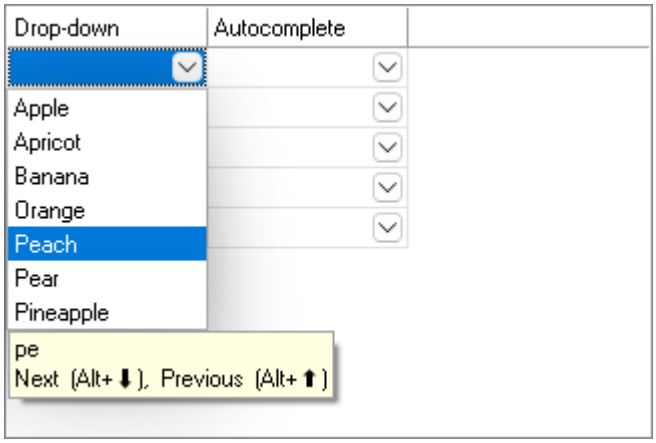
Let's consider a grid with cells linked to a drop-down list containing items Apple, Apricot, Banana, Orange, Peach, Pear, and Pineapple. It can be created with a code snippet placed below. We imply that this snippet is a part of a form also containing an instance of the iGDropDownList control named iGDropDownList1:

```
var items = iGDropDownList1.Items;
items.Add("Apple");
items.Add("Apricot");
items.Add("Banana");
items.Add("Orange");
items.Add("Peach");
items.Add("Pear");
items.Add("Pineapple");

iGrid1.Cols.Add("Drop-down", 100);
iGrid1.Cols.Add("Autocomplete", 100);

iGrid1.Cols[0].CellStyle.DropDownControl = iGDropDownList1;
iGrid1.Cols[1].CellStyle.DropDownControl = iGDropDownList1;

// Set optimal row height to show full size combo buttons
iGrid1.DefaultRow.Height = iGrid1.GetPreferredRowHeight(true, false);

iGrid1.Rows.Count = 5;
```

When we launch this form and open our drop-down list with fruit names in the first cell, we see the following picture:

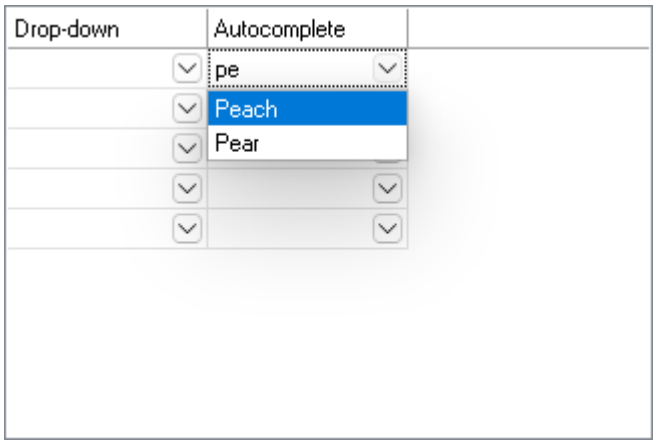


This is the traditional usage mode of drop-down lists — the **drop-down mode**. It supports search-as-type working in seek mode by default. For example, if we type "pe" on the keyboard while the list is opened, the first list item starting with these letters will be highlighted:



As the hint window suggests, you can use the ALT+UP ARROW and ALT+DOWN ARROW keyboard combinations to find all items starting with the typed characters. When you find the required item, you can press ENTER to put it into the cell.

Let's close the list and do another thing. When a cell is selected, type "pe" directly into the cell. We will see that the same drop-down list is opened, but now it contains only the items starting with the typed characters:



We can select the required item with the cursor movement keys, press ENTER, and the chosen item will be saved in the cell. This is the other usage mode of built-in drop-down lists — the **autocomplete mode**. It allows you to use drop-down lists to implement the autocomplete feature for text cells.

Pay attention to the fact that in the second scenario the text editing was activated in the cell — we see the typed characters in it. If we want to ignore the suggestions from the filtered autocomplete list and save the typed characters "as is", we should use the special keyboard combination SHIFT+ENTER for this.

Adjusting the search-as-type functionality for the drop-down and autocomplete modes

The built-in drop-down list implementation, the **iGDropDownList** class, provides you with the **SearchAsType** object property you can use to adjust the search-as-type functionality in drop-down lists. The main sub-properties of this object property are:

PROPERTY	DESCRIPTION
AutoCompleteMode	Specifies the search-as-type mode when the list is used as autocomplete list. Can accept one of the following values from the **iGSearchAsTypeMode** enumeration: **Filter**, **Seek**, or **None**. The default is **Filter**.

PROPERTY	DESCRIPTION
DropDownMode	Specifies the search-as-type mode when the list is used as drop-down list. Has the same **iGSearchAsTypeMode** enumeration type like the **AutoCompleteMode** property. The default is **Filter**.
MatchRule	A property of the **iGMatchRule** enumeration type that specifies the match rule for both autocomplete and drop-down modes. Can be **Contains**, **StartsWith**, or **Custom**. When the **Custom** value is specified, the **SearchAsTypeCustomCompare** event of **iGDropDownList** is used to determine whether an item matches the search criteria. The default is **StartsWith**.

As you can see from the table above, the **AutoCompleteMode** and **DropDownMode** properties allow you to change the default seek and filter search-as-type modes used in drop-down lists shown in drop-down and autocomplete modes respectively.

Built-in drop-down lists display the helper search window with the entered text and keyboard hints like the iGrid control itself. However, the parts of this search window (search text and keyboard hint) may be hidden depending on the context in which the search window is shown:

- List is in drop-down mode and search-as-type is in seek mode — both parts are displayed;
- List is in drop-down mode and search-as-type is in filter mode — only the search text is displayed;
- List is in autocomplete mode and search-as-type is in seek mode — only the keyboard hint is displayed;
- List is in autocomplete mode search-as-type is in filter mode — the search window is not displayed at all.

The rest two Boolean properties of the **SearchAsType** object property, **DisplayKeyboardHintIfNeeded** and **DisplaySearchTextIfNeeded**, allow you to hide the corresponding parts of the search window if required.

Autocomplete functionality without combo buttons

When you assign a drop-down list to the **DropDownControl** property of the cell to use it as autocomplete list, the cell combo button appears automatically. Sometimes, you just need the autocomplete feature without combo buttons. The cell combo button can be removed with the **HideComboButton** flag specified in the **TypeFlags** property of the cells, for example:

```
iGrid1.Cols[0].CellStyle.TypeFlags = iGCellTypeFlags.HideComboButton;
```

The IiGAutoCompleteControl interface

The autocomplete list is assigned to a cell like a drop-down control with the **DropDownControl** property of the cell. To be used for the autocomplete functionality, the object assigned to the **DropDownControl** property must implement the special **IiGAutoCompleteControl** interface.

This interface inherits the **IiGDropDownControl** interface any drop-down control must implement. The **IiGAutoCompleteControl** interface adds some methods to process keyboard input in the cell (like **ProcessKeyPress** and **OnCellTextChange**) and the **ValueSelected** event to inform the cell about selecting a value in the autocomplete list.

The **iGDropDownList** class providing the built-in functionality of drop-down lists also implements the **IiGAutoCompleteControl** interface and thus can be used as an autocomplete control in cells too.

Using different lists for the drop-down and autocomplete functionality

In some cases the autocomplete control (which is shown when editing a cell as text) may differ from the drop-down control (which is shown when the user clicks the combo button or presses the F4 key). For example, you can use a custom cell drop-down editor that does not support the autocomplete functionality. To implement this task, use the **RequestAutoCompleteControl** event of iGrid. This event is raised every time iGrid needs to obtain the autocomplete control for a cell. Set the **Control** argument of this event to the list you want to use as the autocomplete list instead of the attached drop-down control.

25.2.8. Relation Between Combo Box Cell and Its Drop-Down List

When the user selects an item in a drop-down list, the item's value and image index are saved in the **Value** and **ImageIndex** properties of the cell. The drop-down list item itself is saved in the cell's **AuxValue** property.

When you assign a value to a combo box cell from code, iGrid searches for an item with the same value in the drop-down list attached to the cell. If the grid finds such a drop-down list item, it assigns a reference to this item to the **AuxValue** property of the cell, and the image index of the item is assigned to the cell's **ImageIndex** property.

The **AuxValue** property of a cell is changed by iGrid during interactive selection for the combo box cells only, but for cells of other types you can use it for your own needs similar to the Tag property in visual controls. When you modify the **AuxValue** property from code, iGrid doesn't modify the **Value** and **ImageIndex** properties of the cell.

If you want to specify a default value for combo box cells in a column, you should specify both the value itself and the drop-down list item associated with it as the auxiliary cell value. The example below demonstrates how to initialize the combo box cells in the first column with the second item from the attached drop-down list:

```
iGDropDownList fDropDownList = new iGDropDownList();

private void Form1_Load(object sender, EventArgs e)
{
    fDropDownList.Items.Add("Item 1");
    fDropDownList.Items.Add("Item 2");
    fDropDownList.Items.Add("Item 3");

    iGrid1.Cols.Count = 3;

    iGrid1.Cols[0].CellStyle.DropDownControl = fDropDownList;
    iGrid1.Cols[0].DefaultCellValue = fDropDownList.Items[1].Value;
    iGrid1.Cols[0].DefaultCellAuxValue = fDropDownList.Items[1];

    iGrid1.Rows.Count = 5;
}
```

By default iGrid obtains the text displayed in a combo box cell from the **Text** property of the drop-down list item stored in the **AuxValue** property of the cell. You can change this behavior by specifying the **ComboPreferValue** flag in the **TypeFlags** property of the cell. If this flag is specified, the text displayed in the cell is obtained from the cell's value. In this case you can also apply various display formats to the cell value with the **FormatString** and **FormatProvider** properties of the cell.

If a combo box cell can be edited as text, iGrid searches for the drop-down list item that corresponds to the entered text after editing has been finished. By default iGrid searches for the first item that has the same text, and if such an item is found, it is saved to the **AuxValue** property of the cell and is used to display the cell's text. If the **ComboPreferValue** flag is specified for the cell, iGrid first

converts the entered text to the type of the cell value, and then seeks the drop-down list item with this value.

If the user selected a drop-down list item in a cell and you remove this item from the drop-down list, the reference to this item (an instance of the **iGDropDownListItem** class) is still stored in the **AuxValue** property of the cell. This means that the cell will display the text from the selected combo list item until you assign null (Nothing in VB) to this property. This feature is especially useful if you need to use the same drop-down list in several cells, but the contents of the list may change. For example, in some scenarios a drop-down list item can be selected only once and it should be removed from the list after it has been selected in a cell. In this case the item will be no longer available in the list, but it still will be present in the cell in which it has been selected.

If you use objects as values of drop-down list items, read also the advanced Objects as Drop-Down List Item Values topic to know how to speed up iGrid performance for long drop-down lists.

25.3. Check Box Cells

25.3.1. Creating and Formatting Check Box Cells

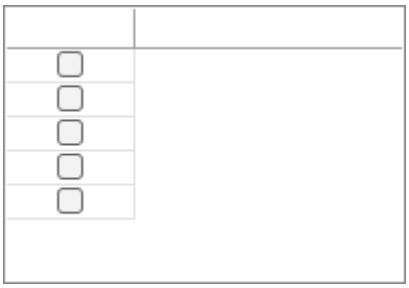
To create a check box cell, you just need to specify the **Check** style in the **Type** property of the cell.

The position of the check box control inside a check box cell can be adjusted horizontally and vertically. By default, it is aligned top-left in the cell, but this position can be changed through the **ImageAlign** property of the cell (to save some memory, we use the existing property of the cell because it is not used in this situation as the iGrid check box cells cannot have other items such as text or image).

Thus, in a typical situation, if you need to create a check box column with the check boxes centered horizontally, your code will look similar to this:

```
iGCol myCol = iGrid1.Cols.Add();
myCol.CellStyle.Type = iGCellType.Check;
myCol.CellStyle.ImageAlign = iGContentAlignment.TopCenter;
iGrid1.Rows.Count = 5;
```

This code gives us the following picture:



25.3.2. Check Box States and Cell Values

You can use the two-state and three-state check box controls in iGrid check box cells. The **TypeFlags** property of a cell or its style object allows you to specify whether the check box control will have the two or three states. By default the check box control has two states — unchecked and checked. If you specify the **iGCellTypeFlags**.**CheckThreeState** flag in the **TypeFlags** property of a cell, the check box in this cell can be in 3 states — unchecked, checked, and indeterminate.

The state of the check box control in a check box cell is linked to the cell value. When you change the check box state interactively, iGrid changes the linked value of the check box cell immediately, and vice versa. The table below explains how iGrid determines the state of the check box based on the cell value and how the new cell value is set after changing the check box state:

CURRENT VALUE	VISUAL STATE OF 2-STATE CHECK BOX	VISUAL STATE OF 3-STATE CHECK BOX	NEW VALUE OF 2-STATE CHECK BOX	NEW VALUE OF 3-STATE CHECK BOX
True	checked	checked	False	**Indeterminate**
False	unchecked	unchecked	True	**Checked**
Unchecked	unchecked	unchecked	**Checked**	**Checked**
Checked	checked	checked	**Unchecked**	**Indeterminate**
Indeterminate	checked	indeterminate	**Unchecked**	**Unchecked**
0	unchecked	unchecked	1	1
1	checked	checked	0	2
2	checked	indeterminate	0	0
any other numeric value	checked	checked	0	2
null (Nothing in VB)	unchecked	unchecked	True	**Checked**
the zero-length string	unchecked	unchecked	True	**Checked**
other values	checked	checked	False	**Indeterminate**

In the table above:

- True and False are the .NET **Boolean** values.
- The **Checked**, **Unchecked** and **Indeterminate** values are the values of the .NET **CheckState** enumeration from the **System.Windows.Forms** namespace.

26. CUSTOM EDITING

26.1. Custom Cell Editors

26.1.1. Custom Cell Editor Model

The iGrid custom cell editor model is based on the abstract **iGCellEditorBase** class. To implement a custom cell editor, inherit this class and implement its two mandatory members — the **EditControl** property and the **SetBounds** method. The custom edit control should be a descendant of the standard System.Windows.Forms.**Control** class, and its instance is returned from the **EditControl** property. The **SetBounds** method is called by iGrid when a cell is put in edit mode so you can position the custom edit control in the cell properly.

To specify a custom cell editor for a cell or set of cells, assign a reference to the custom cell editor object to the **CustomEditor** property of the cell object (**iGCell**) or cell style object (**iGCellStyle**). The **CustomEditor** property has the **iGCellEditorBase** type and accepts any objects based on this class. Below is an example of how to use a NumericUpDownCellEditor class as the editor for the cells in the first column:

```
iGrid1.Cols[0].CellStyle.CustomEditor = new NumericUpDownCellEditor();
```

The NumericUpDownCellEditor class that represents a custom editor based on the standard Windows Forms **NumericUpDown** control might look like this:

```
internal class NumericUpDownCellEditor : iGCellEditorBase
{
  private NumericUpDown fNumericUpDown = new NumericUpDown() {
    BorderStyle = BorderStyle.None, Maximum = 10000, Increment = 5};

  public override Control EditControl
  {
    get
    {
       return fNumericUpDown;
    }
  }

  public override void SetBounds(
    Rectangle textBounds, Rectangle suggestedBounds)
  {
    fNumericUpDown.Bounds = suggestedBounds;
  }
}
```

iGrid automatically sets the corresponding colors for your custom editor, processes general control keys as TAB or ENTER in it, etc. For more information about this process, see the Details about Custom Editing topic in this manual.

Note that you can create just one instance of the class that implements custom editing and use it in several columns of the same grid, or in several grids on the form or in the whole project. In this case all custom editing functionality is concentrated in one class which organizes your code well. If your custom editing class encapsulates a functionality that uses a lot of resources (for instance, a dictionary in a spell-checking system), the approach with one custom editor object will also save computer resources significantly.

26.1.2. Details about Custom Editing

Used terms

For better explanation, let's introduce two terms. When we write "custom cell editor object", we mean an instance of the custom cell editor class which inherits **iGCellEditorBase** and is stored in the **CustomEditor** property of the corresponding cell. If we write "custom edit control" or simply "edit control", we mean the visual control users work with when they are editing a cell — i.e. what is returned from the **EditControl** property of the custom cell editor object. If you see the term "custom cell editor" or its shorter form "custom editor", then the general custom editing construction is implied and there is no need to distinguish its constituent part.

Passing cell values to custom editors

The first question you should answer when you implement a custom cell editor is how to pass the value of the edited cell to the custom editor when editing is started and back to the grid when editing is finished. iGrid provides you with two approaches, and the used way is determined by the Boolean value returned from the **PassValueAsText** property of the **iGCellEditorBase** descendant.

The first approach is to pass the cell value as text to the custom edit control. The text representation of the cell value, or the cell text, is passed through the **Text** property of the edit control (every edit control implements this property as the edit control is derived from the standard Windows Forms **Control** class). To use this approach, return True from the **PassValueAsText** property of your custom cell editor object. You can also omit this property implementation in your class as **PassValueAsText** in its default implementation in **iGCellEditorBase** returns True.

The alternative approach is to pass the cell value "as is" or even convert it using a special algorithm before it appears in the custom edit control. For that, redefine **PassValueAsText** in your class and return False from it. In this case iGrid will use the **Value** property of your custom cell editor object to pass the cell value between the cell and your custom editor. In fact, you must implement the **Value** property in your class if you return False from **PassValueAsText** as otherwise iGrid has no way to pass the cell value to the custom editor.

One of the good examples of the second approach is a custom cell editor based on the standard **DateTimePicker** control. This control can accept date values through both **Text** and **Value** properties, but the latter one is much more preferable. If we pass a date value using the **Text** property, a **DateTimePicker** tries to parse text as date using the available regional date formats and in the general case this process may fail if the date string isn't recognized. But it never fails if we pass the date value in its native format through the **Value** property. Here is the corresponding example of the editor based on the standard **DateTimePicker** control:

```
internal class DateTimePickerCellEditor : iGCellEditorBase
{
  // The edit control itself
  private DateTimePicker fDateTimePicker = new DateTimePicker()

  // Obligatory members
  public override Control EditControl
  {
    get { return fDateTimePicker; }
  }
  public override void SetBounds(
    Rectangle textBounds, Rectangle suggestedBounds)
  {
    fDateTimePicker.Bounds = textBounds;
  }

  // Editing cell value 'as is' in its native format
  public override bool PassValueAsText
  {
    get { return false; }
  }
  public override object Value
  {
    get { return fDateTimePicker.Value; }
    set { fDateTimePicker.Value = (DateTime)value; }
  }
}
```

When the user commits editing, iGrid reads either the **Text** property of the edit control or the **Value** property of the custom cell editor object to get the new value depending on **PassValueAsText** the same way.

Custom editor properties set on editing

When a cell is put into edit mode, iGrid does the following with the custom cell editor object:

1. The edit control is hidden by setting its **Visible** property to False before iGrid will adjust other properties of the control.
2. iGrid sets the **BackColor**, **ForeColor** and **Font** properties of the custom edit control so it uses the same colors and font like the edited cell.
3. The corresponding cell settings are assigned to the following properties of the custom cell editor object: **MaxInputLength**, **TextAlign**, **LineAlign**, **WordWrap**, **Multiline**, **PasswordChar**. These properties are virtual (Overridable in VB), so they may be implemented in the custom editor class only if needed and if the edit control supports them.
4. iGrid calls the **SetBounds** method of the custom cell editor object so you can position the edit control as required.
5. If the cell value should be passed as text to the edit control (**PassValueAsText** returns True), the corresponding text is assigned to the **Text** property of the edit control. Otherwise, the cell value is assigned "as is" to the **Value** property of the custom cell editor object.
6. Depending on the action that caused editing (key press, double-click, etc.), iGrid automatically selects the text that is put into the edit control. To do that, iGrid assigns the corresponding values to the **SelectionStart** and **SelectionLength** properties of the custom cell editor object. These Selection* properties are virtual, so your custom cell editor class may not contain their implementation if the used edit control does not support the control over selection. Note that this makes sense only if the cell value is passed as text — otherwise

iGrid simply does not have any information about the text representation of the value in the edit control.

7. The edit control is activated for editing. First, its **CausesValidation** property is set to True (required by the internal editing infrastructure). Then the edit control is displayed (its **Visible** property is set to True) and input focus is moved to it by calling the edit control's **Focus** method.

The parameters of the SetBounds method

iGrid provides you with two rectangles in the parameters of the **SetBounds** method. The first of them, **textBounds**, contains the whole rectangle available for displaying cell text. The second rectangle, **suggestedBounds**, is used if the cell text is displayed on one line. The **suggestedBounds** rectangle is calculated taking into account the vertical alignment of the cell text. This rectangle is very helpful if you need to align cell text vertically in the whole available rectangle but your edit control does not have a built-in option for that (like the standard **TextBox** control).

Note that **suggestedBounds** differs from **textBounds** only if the cell text is displayed on one line. Generally this happens if the word wrapping and multiline editing options are not set for the cell (see the **TextFormatFlags** and **TypeFlags** properties of **iGCell**). Otherwise both rectangles are equal.

You decide which rectangle to use for positioning your custom edit control. This may depend on the resize capabilities of the custom edit control and other factors.

General editing event model and custom editing

As you may know from the Common Events Related to Editing topic of this manual, iGrid provides you with some general events fired when the user starts and finishes editing of any cell. All these events also work for custom cell editors based on the **iGCellEditorBase** class.

iGrid's text box and combo box cells also support the specific set of TextBox* events like **TextBoxFilterChar** and **TextBoxMouseDown** (for more information, see the Text Edit Events and the TextBox Property topic). If your custom edit control properly supports general text, keyboard and mouse events (**TextChanged**, **KeyPress**, **MouseDown**, etc.), all the related iGrid TextBox* events will work for custom cell editors as well. And even iGrid's **TextBox** object property can be used to access retrieve the corresponding information of the current edited text and its selection if these parts are supported in your custom editor.

26.1.3. Non-iGCellEditorBase-based Approach

The custom editing model based on the **iGCellEditorBase** class works perfectly and provides you with all the standard functionality which integrates an external control as an iGrid cell editor seamlessly. However, if you need sophisticated tuning for the custom editing process, you may use other members of iGrid to manually place the custom control over the edited iGrid cell and activate it, process special control keystrokes like the TAB or SHIFT+ENTER keys in it, etc.

The remainder of this topic lists the corresponding iGrid members you can use for that. In any case, we strongly recommend that you test your custom editors implemented this way thoroughly in different scenarios (using the keyboard, mouse, etc.) as you may face a series of problems you will need to solve in your code which you never get if you use the **iGCellEditorBase** descendants. Moreover, these points should be tested every time when you upgrade to a newer version of iGrid — while in the case of **iGCellEditorBase** all required functionality is updated automatically inside iGrid as a part of its internal infrastructure.

The following two events should be the core part of your custom editing functionality:

- **RequestEdit**. This event helps you to determine the time to begin the editing. Simply set the **DoDefault** field of its arguments to False to prevent built-in editing and place your own control over cell and activate it from this event.

- **QuitCustomEdit**. This event helps you to determine the time to cancel editing. Generally it is fired in the following typical cases when editing should be cancelled: ESC is pressed, the grid is scrolled or its size is changed, a row or column is added or removed, etc.

Other useful iGrid members to get the best view and behavior when you implement custom editing are as follows:

- Use the **MouseDownLocked** property to prevent selecting other cells, sorting and other actions related to mouse events. It is helpful if you need to validate the new cell value and you wish to prevent the user from selecting any cell in the grid while the new value is invalid.
- Use the **HotTracking** property to prohibit hot tracking effects in the grid's cells. It is useful when you implement custom editing. If an item indicates its hot state, it means that it can be changed with the single mouse click that may be impossible when a cell is edited.
- Use the **DrawAsFocused** property to draw iGrid as if it had input focus. When you show your own edit control in a cell, iGrid loses input focus and its selected cells are filled with the **SelCellsBackColorNoFocus** color. To prevent this, set the **DrawAsFocused** property to True.
- The **CurCellComboPressed** property allows you to make the combo button in the current cell pressed. This property is intended to be used to implement custom drop-down editing. When you show a drop-down list for a cell when the user clicks the combo button in the cell, set this property to True to display the combo button in pressed state while the list is on the screen. After the drop-down list is hidden, set this property to False. The fact is that generally you display a drop-down list in custom editing from the **RequestEdit** event, but the combo button does not remain pressed after this event and you need a way to draw it pressed while your list is on the screen. The **CurCellComboPressed** property does this work for you.

26.2. Custom Drop-Down Controls

26.2.1. The IiGDropDownControl Interface

iGrid allows you to create custom drop-down controls which can be used to edit cells. To create a custom drop-down control, implement the **IiGDropDownControl** interface in it. The reminder of this topic explains the main concept of how iGrid uses your own drop-down control based on the **IiGDropDownControl** interface.

iGrid manages a drop-down control by itself. It gets the control returned by the **GetDropDownControl** method, and displays it in its own drop-down form. In fact, iGrid incorporates your drop-down control into this drop-down form as a child control.

The width and height of the drop-down form depends on the **Height** and **Width** properties. If the **Width** property returns a value greater or equal to zero, the width of the drop-down form's client area will be equal to it. Otherwise, the width of the drop-down form will be equal to the cell with. The **Height** property should return the height of the drop-down control.

Note that the **Width** and **Height** properties must return the width and height of your drop-down control but not the drop-down form it is placed in. The drop-down form can have its own border, and the values of these properties are not equal to the width and height of the drop-down form in the general case. iGrid automatically resizes the drop-down form taking account its border to display your drop-down control with the size specified in the **Width** and **Height** properties.

iGrid requests the width and height of a drop-down control just before it is shown. If you need to update its size after it has been shown, you should invoke the **UpdateDropDownLocationAndSize** method of iGrid. When this method is invoked, iGrid accesses the **Width** and **Height** properties of the drop-down control and adjusts the size and location of the drop-down form based on the values returned by these properties.

The drop-down form can have both the sizeable and fixed border. The border style depends on the **Sizeable** property of the interface. If this property returns True, iGrid allows the user to resize the drop-down form by dragging its border as you can do it with any resizable window in Windows.

When the drop-down form is about to be shown, iGrid requests the size (width and height) of the drop-down control and resizes the drop-down form to fit the control's size. When the size is set, iGrid sets the value stored in the **AuxValue** property of the cell to the drop-down control using the **SelectedItem** property.

The drop-down control can be hidden with the **CommitEditCurCell** method of iGrid. Invoke this method when the user selects a value in the drop-down control, and the value should be saved to the cell. Invoke the **CancelEditCurCell** method to cancel editing without saving the entered value. iGrid can also hide the drop-down control automatically. It may happen when the user activates another application or clicks on the area out of the drop-down form. In this case the value returned by the **CommitOnHide** property indicates whether to save the entered value. If it returns True, the value will be saved; otherwise, canceled.

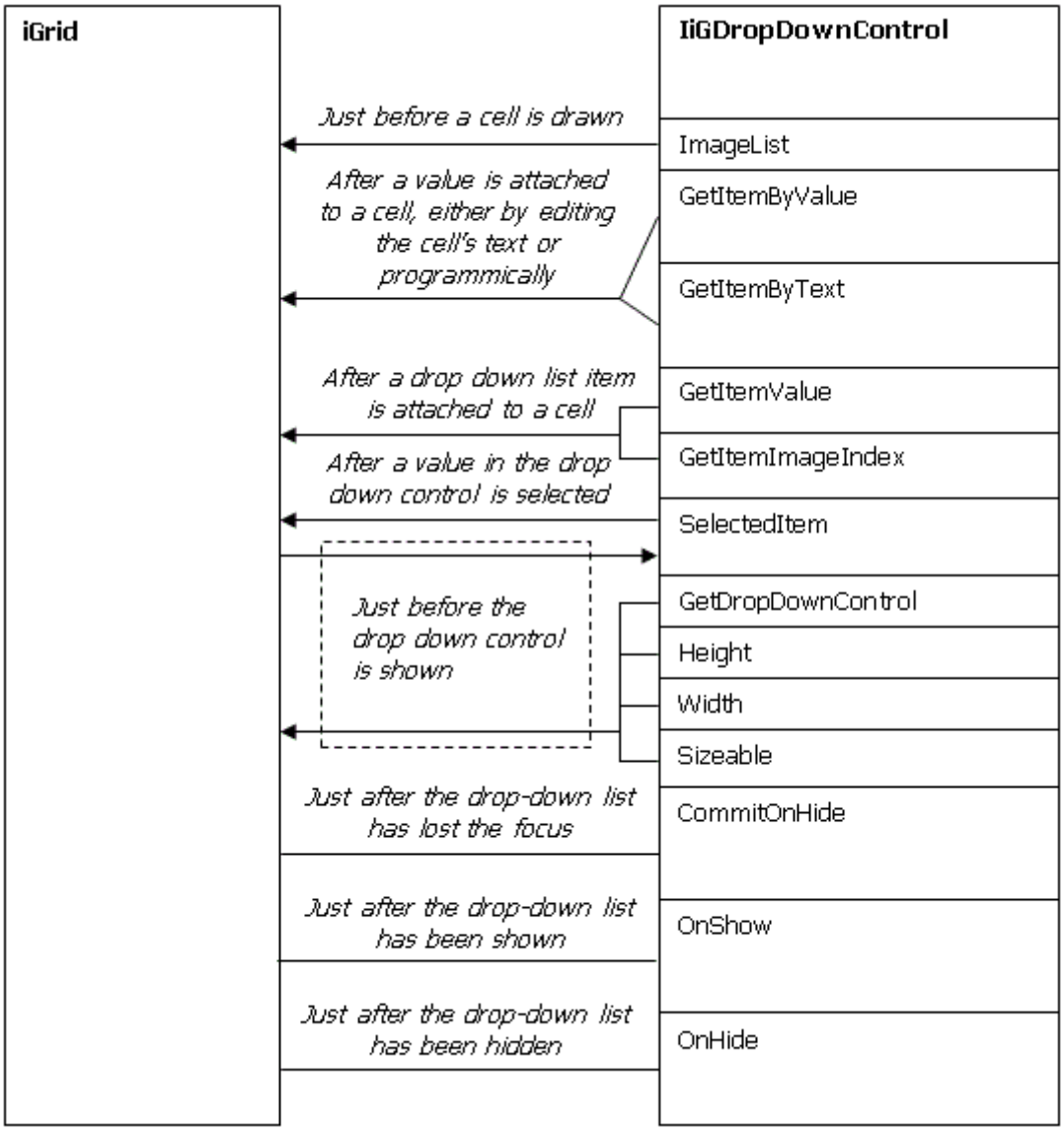
Note that to call the **CommitEditCurCell** or **CancelEditCurCell** method of iGrid, you need a reference to an instance of the iGrid class in your drop-down control. The reference to iGrid that displays your drop-down control is passed as the first parameter of the **GetDropDownControl** method of the **IiGDropDownControl** interface, and you need to store it internally in your drop-down control to access it later.

After editing is committed, iGrid assigns the value returned by the **SelectedItem** property of the drop-down control to the **AuxValue** property of the cell. The cell's value and image index are set to the values returned by the **GetItemValue** and **GetItemImageIndex** properties of the drop-down control respectively.

When a cell (which the drop-down control is attached to) is displayed, its text is obtained from the **ToString** method of the drop-down control item stored in the **AuxValue** property (unless the **ComboPreferValue** flag is specified).

When you set a cell's value programmatically, iGrid automatically sets up the **AuxValue** property of the cell. The value assigned to the **AuxValue** property is the drop-down control item obtained with the **GetItemByValue** method of the drop-down control. The **GetItemByValue** method is also used after the cell is edited as text and the **ComboPreferValue** flag is specified. If the **ComboPreferValue** flag is not specified, the **GetItemByText** method is used.

The following scheme shows the main relations between iGrid and the implemented members of the **IiGDropDownControl** interface:



Examine also all the members of the **IiGDropDownControl** interface to find out more about other features you can use.

26.2.2. Custom Autocomplete Controls

In addition to the standard implementation of the autocomplete functionality (**iGDropDownList**), you can create your own autocomplete control. To do it, you should implement the **IiGAutoCompleteControl** interface. This interface is derived from the **IiGDropDownControl** interface and has the following members:

- The **OnCellTextChange** method — this method is invoked by iGrid when the text in a cell is changed while editing this cell as text. Its only parameter, which is named text, provides you with the text entered to the cell. This method should return a Boolean value indicating whether the autocomplete control has items which match the entered text and it should be displayed.
- The **ProcessKeyDown** method — processes the key down event when a cell is being edited as text. This method is invoked when the control which is used to edit the cell (a text box) receives the **KeyDown** event. The arguments of this event are passed in the e parameter of the method. If the autocomplete control has handled this event and wants to prevent the cell editor from handling the key, you should set the **Handled** parameter of the event arguments to True.

- The **ProcessKeyPress** method — processes the key press event when a cell is being edited as text. This method is invoked when the control which is used to edit the cell (a text box) receives the **KeyPress** event. The arguments of this event are passed in the e parameter of the method. If the autocomplete control has handled this event and wants to prevent the cell editor from handling the key, you should set the **Handled** parameter of the event arguments to True.
- The **ProcessKeyUp** method — processes the key up event when a cell is being edited as text. This method is invoked when the control which is used to edit the cell (a text box) receives the **KeyUp** event. The arguments of this event are passed in the e parameter of the method. If the autocomplete control has handled this event and wants to prevent the cell editor from handling the key, you should set the **Handled** parameter of the event arguments to True.
- The **ValueSelected** event — should be raised when a value in the autocomplete control is selected and editing should be finished.

27. CELL ELLIPSIS BUTTONS

27.1. Cell Ellipsis Button Basics

You can add an ellipsis button to any cell of iGrid.NET. As a rule, this button is used to open a custom cell editor or display detailed info for the cell.

The ellipsis button can be displayed as an optional element in a cell of any type:

TEXT CELL	CHECK BOX CELL	COMBO BOX CELL
		

To show the ellipsis button in a cell, specify the **HasEllipsisButton** flag in the **TypeFlags** property of the cell or a style attached to the cell. The following example demonstrates how to show ellipsis buttons in the cells of the first column:

```
iGrid1.Cols[0].CellStyle.TypeFlags = iGCellTypeFlags.HasEllipsisButton;
```

As well as combo buttons, ellipsis buttons can be shown in all cells or only in the current cell. If the **ShowControlsInAllCells** property of iGrid is set to True (the default value), the ellipsis buttons are displayed in all the cells. If **ShowControlsInAllCells** equals False, the ellipsis button is shown only in the current cell.

When an ellipsis button is clicked, iGrid raises the **CellEllipsisButtonClick** event. Use this event to perform your action when an ellipsis button is clicked (for instance, to show a custom dialog box).

Pay attention to the fact that the **RequestEdit** event is not raised when an ellipsis button is clicked; this event is used to control text editing which is not the same as cell ellipsis button click. To disable text editing in a text cell with an ellipsis button, you can use the **RequestEdit** event or specify the **NoTextEdit** flag in the **TypeFlags** property of the cell or style attached to it. Setting the cell's **ReadOnly** property to True completely disables any interactive activity in the cell except the ability to select it, and this also hides the cell's ellipsis button.

Note that if you show a dialog when an ellipsis button is clicked, the grid loses input focus and the current cell is drawn as inactive (with the different background color, without the focus rectangle, etc.). To prevent this, use the **DrawAsFocused** property of iGrid. The following example demonstrates how to force a grid to draw itself as focused while a dialog is shown and the grid has temporarily lost input focus:

```
private void iGrid1_EllipsisButtonClick(
  object sender, iGridLib.iGEllipsisButtonClickEventArgs e)
{
  iGrid1.DrawAsFocused = true;
  ... //Show the custom dialog
  iGrid1.DrawAsFocused = false;
}
```

27.2. Customizing Ellipsis Button Appearance

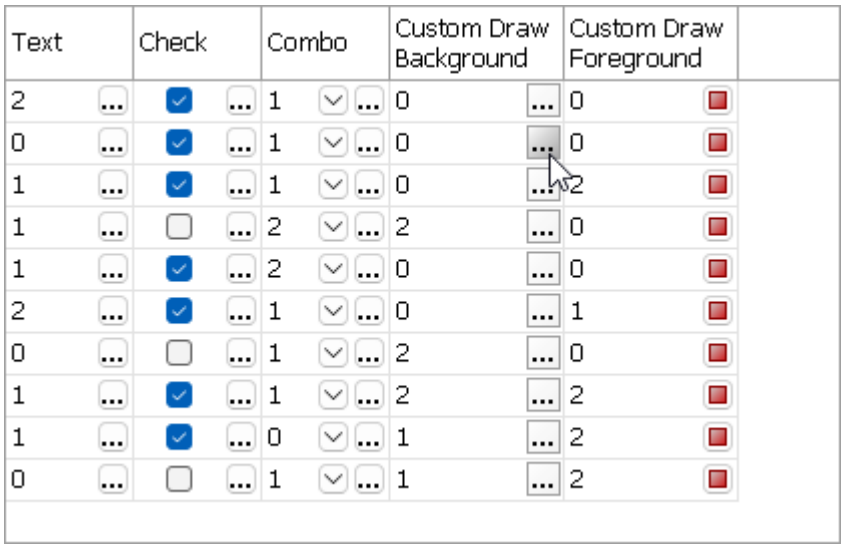
The default glyph displayed on cell ellipsis buttons is ellipsis. You can specify your own image for all ellipsis buttons with the **EllipsisButtonGlyph** property of iGrid. This property of the **Image** type accepts any image you can upload with .NET methods, for example:

```
iGrid1.EllipsisButtonGlyph = Image.FromFile(@"C:\Pictures\Magnifier.png");
```

Below is an example of the ellipsis button with a custom image:

You can also draw the background and/or foreground of the ellipsis button from scratch using custom drawing. The **CustomDrawCellEllipsisButtonForeground** and **CustomDrawCellEllipsisButtonBackground** events are used for this purpose. When you handle these events, also set the **DoDefault** field of the event arguments to False to prevent iGrid from drawing the standard contents.

The custom draw feature can be used to change the ellipsis glyph only in some ellipsis buttons — for example, like in the buttons inside the cells of the last column on the following screenshot:



This task can be implemented with the **CustomDrawCellEllipsisButtonForeground** event:

```
private void fGrid_CustomDrawCellEllipsisButtonForeground(
  object sender, iGCustomDrawEllipsisButtonEventArgs e)
{
  if (e.ColIndex == fGrid.Cols.Count - 1)
  {
    Rectangle myBounds = e.Bounds;
    myBounds.Inflate(-4, -4);

    if (myBounds.Width > 0 && myBounds.Height > 0)
    {
      e.Graphics.DrawImageUnscaledAndClipped(fPictureBox.Image, myBounds);
    }

    e.DoDefault = false;
  }
}
```

We can also redefine only the drawing of ellipsis button background and use the default ellipsis glyph like in the Custom Draw Background column on the screenshot above:

```
private void fGrid_CustomDrawCellEllipsisButtonBackground(
  object sender, iGCustomDrawEllipsisButtonEventArgs e)
{
  if (e.ColIndex == fGrid.Cols.Count - 2)
  {
    // Determine the colors of the background
    Color myColor1, myColor2;
    switch (e.State)
    {
      case iGControlState.Pressed:
        myColor1 = SystemColors.ControlDark;
        myColor2 = SystemColors.ControlLightLight;
        break;
      case iGControlState.Hot:
        myColor1 = SystemColors.ControlLightLight;
        myColor2 = SystemColors.ControlDark;
        break;
      default:
        myColor1 = SystemColors.ControlLightLight;
        myColor2 = SystemColors.Control;
        break;
    }

    // Draw the background
    using (LinearGradientBrush myBrush = new LinearGradientBrush(
      e.Bounds, myColor1, myColor2, 45))
    {
      e.Graphics.FillRectangle(myBrush, e.Bounds);
    }
    e.Graphics.DrawRectangle(SystemPens.ControlDark,
      e.Bounds.X, e.Bounds.Y, e.Bounds.Width - 1, e.Bounds.Height - 1);

    // Notify the grid that the background has been drawn
    // and there is no need to draw it
    e.DoDefault = false;
  }
}
```

Notice that the cell ellipsis button under the mouse pointer on the screenshot above is highlighted using the custom hot effect we coded ourselves in the event handler.

27.3. Tooltips for Cell Ellipsis Buttons

Cell ellipsis buttons do not display tooltips by default. If you want to display a tooltip explaining what an ellipsis button does or show other useful information related to the ellipsis button under the mouse pointer, you can generate the tooltip text on the fly with the **RequestCellElemControlToolTipText** event. Below is an example showing how to display a tooltip with the row and column index of the cell in which the ellipsis button under the mouse pointer is placed:

```
private void iGrid1_RequestCellElemControlToolTipText(
  object sender, iGRequestElemControlToolTipTextEventArgs e)
{
  if (e.ElemControl == iGElemControl.EllipsisButton)
  {
    e.Text = $"Row: {e.RowIndex} | Column: {e.ColIndex}";
  }
}
```

28. WORKING WITH DATABASES

28.1. Viewing ADO.NET Data in iGrid

28.1.1. Populating iGrid with Data

iGrid implements the **FillWithData** method you can use to populate it with the data from one of the following ADO.NET data sources:

- a **DataTable** object;
- a **DataView** object;
- any object that implements the **IDbCommand** interface (**OleDbCommand**, **SqlCommand**, etc.);
- any object that implements the **IDataReader** interface (**OleDbDataReader**, **SqlDataReader**, etc.).

There are several overloaded versions of the **FillWithData** method to provide various options to configure the data population process. But all versions of the method accept a data source object as the first parameter. In the simplest case you can fill iGrid with the data from a data source like **DataTable** in one statement using the **FillWithData** version with one parameter:

```
iGrid1.FillWithData(DataTable1);
```

This call of the **FillWithData** method clears the contents of iGrid, creates the required set of columns to display all the fields from the specified data source, and finally copies the data into iGrid rows.

Unless you specify otherwise with method parameters, the **FillWithData** method automatically creates grid columns corresponding to the columns in the specified data source. To do this, the method sets column properties according to the following rules:

- **Text** — the caption of the data source's column; if the caption isn't specified, it is the name of the column.
- **Key** — the name of the column in the data source.
- **CellStyle**.**Type** — depends on the type of the column in the data source. If the data source's column has the Boolean type, the column in iGrid displays check box cells; in other cases normal text box cells are used.
- **CellStyle**.**ImageAlign** — if the data source's column has the **Boolean** type, the column's cell style object uses the **TopCenter** image alignment; **TopLeft** is used otherwise.
- **CellStyle**.**TextAlign** — If the data source's column has a numeric type, the column's cell style object uses the **TopRight** text alignment; **TopLeft** is used otherwise.
- **DefaultCellValue** — the default value of the data source's column if it exists, null (Nothing in VB) otherwise.
- **CellStyle**.**ReadOnly** — depends on the read only property of the data source column.
- **CellStyle**.**ValueType** — the data type of the column.

The **FillWithData** method provides you with an ability to load not all the columns from the specified data source to the grid. To do this, you need to create the required set of columns in the grid first and then invoke the corresponding overloaded version of the method with the parameter **useCurColSet** set to True. The column keys of the existing columns in iGrid will be used to link them to the fields in the specified data source. If an iGrid column does not have a string key or it does not correspond to any column in the data source, the cells in this column will be empty.

Below is an example demonstrating how to display only the ID and Name columns from a DataTable in iGrid:

```
iGrid1.Cols.Add("id", "ID");
iGrid1.Cols.Add("name", "Name");
iGrid1.FillWithData(DataTable1, true);
```

In the code above, we used one of the overloaded versions of the **Add** method of the **Cols** collection of iGrid to create columns and specify their keys. The **Add** call above accepts two arguments: column key and column title (what the user sees in the column header). The column keys written in lowercase, "id" and "name", will be used by the **FillWithData** method to find the corresponding fields in the DataTable to show them in iGrid.

When the **FillWithData** method creates rows for the data, iGrid memory pages are used to allocate memory. If the provided data sources has thousands of rows, you can get some performance gain by adjusting the value of the **PageCapacity** property of iGrid before calling the method. To find out more about iGrid memory pages, read the Allocating Memory for iGrid Cells topic.

Pay attention to the fact that even if iGrid is editable, changes in its cells are not reflected in the data source you used to populate iGrid. The **FillWithData** method only copies the data into iGrid, and you need to write some additional code if you want to update the data source accordingly to your data update strategy. One of the numerous possible techniques is considered in the Data Binding Sample for DataTable section in this manual.

Finally, here are some notes about data sources used in the **FillWithData** method. If you use a data reader (an object implementing the **IDataReader** interface), by default it is automatically closed after populating iGrid with any version of the **FillWithData** method. See the Data Population Options topic to know how to specify the option not to close the data reader in this case.

When you pass an ADO.NET command object (an object implementing the **IDbCommand** interface) as the data source to the **FillWithData** method, the related connection isn't automatically opened — you must open it before the call manually. The following example demonstrates a right way to fill iGrid from an **OleDbCommand** object:

```
// Creating data connection and SQL command
OleDbConnection myConnection = new OleDbConnection();
myConnection.ConnectionString = @"Provider=Microsoft.Jet.OLEDB.4.0;" +
  @"Data Source=" + Environment.CurrentDirectory + @"\Test.mdb;";
OleDbCommand myCommand = new OleDbCommand();
myCommand.Connection = myConnection;
myCommand.CommandText = "Select * From Customers";

// Populating iGrid with the data
myConnection.Open();
iGrid1.FillWithData(myCommand);
myConnection.Close()
```

28.1.2. Data Population Options

The **FillWithData** method used to populate iGrid with data has many overloaded versions providing you with the ability to specify various options — whether to use the current column set, which column should be used as a provider of row keys, the maximal number of records to copy to iGrid, etc. There are dozens of combinations of these options you could use in a particular call, and it does not make sense to create overloaded versions of the method for all possible parameter combinations just to provide you with the ability to specify only the desired set of parameters. A universal version of the **FillWithData** method that would provide parameters to set all possible options is also not a good solution because in the vast majority of cases you would think what "empty" or "NotSet" values you need to specify for unused options in such a call. iGrid solves this problem by providing the **iGFillWithDataOptions** class and the overloaded version of the **FillWithData** method that accepts only a data set and an instance of this class:

```
public void FillWithData(object dataSource, iGFillWithDataOptions options)
```

The **iGFillWithDataOptions** class is used to group all **FillWithData** options in one object. It is defined as follows:

```
public class iGFillWithDataOptions
{
  public bool   UseCurColSet    = false;
  public string RowKeyCol       = null;
  public bool   AddRowKeyCol    = false;
  public string RowLevelCol     = null;
  public bool   AddRowLevelCol  = false;
  public bool   AddTreeButtons  = true;
  public int    DataStartRow    = 0;
  public int    DataRowCount    = int.MaxValue;
  public bool   CloseDataReader = true;
  public bool   AppendRows      = false;
}
```

The key point is that when you create an instance of **iGFillWithDataOptions**, all its properties representing the **FillWithData** options are set to their default values, and you need to specify only non-default options. Let's consider an example that demonstrates the benefits of this approach.

If you want to show only the first 10 rows of a data table in iGrid and want to use the values of the ID column as row keys, you could use the following overloaded version of **FillWithData** in which options are specified as simple parameters:

```
public void FillWithData(
  object dataSource, bool useCurColSet,
  string rowKeyCol, bool addRowKeyCol,
  string rowLevelCol, bool addRowLevelCol, bool addTreeButtons,
  int dataStartRow, int dataRowCount)
```

The call would look like this:

```
iGrid1.FillWithData(dataTable, false, "ID", false, null, false, false, 0,
10);
```

Now look at a more elegant and self-descriptive solution based on **iGFillWithDataOptions**:

```
var opts = new iGFillWithDataOptions()
  { RowKeyCol = "ID", DataRowCount = 10 };
iGrid1.FillWithData(dataTable, opts);
```

Most field names of the **iGFillWithDataOptions** class correspond to the parameters of existing overloaded versions of the **FillWithData** method. The remaining fields provide the following features not available in any of the **FillWithData** overloads:

- The **CloseDataReader** field indicates whether an **IDataReader**-based data source must be closed automatically after uploading data into iGrid. The value of this field is not used for other kinds of data sources.
- The **AppendRows** field controls how new rows are added from the data source. When set to True, these rows are appended to the existing rows. If set to False, they replace the current rows. In append mode, iGrid keeps using the current set of columns, so the **UseCurColSet** setting is ignored.

There is one more benefit provided by the approach with the **iGFillWithDataOptions** class. If you have several calls of the **FillWithData** method that use the same arguments, the **iGFillWithDataOptions** class allows you to encapsulate them in one object and use it in all similar calls to **FillWithData**.

28.1.3. The FillWithDataRowAdded Event

iGrid includes the **FillWithDataRowAdded** event, which works alongside the **FillWithData** method. This event is triggered each time a new row is added from the data set during the fill process. You can use it to track loading progress — for example, by displaying a progress indicator when working with large data sources — or to initialize row properties using values from the added data row.

As an example, let's imagine a grid that will be populated with the data from a data table. We want to save the original value of the Country field in the **Tag** property of rows. If we use the **FillWithData** method to populate iGrid, the following event handler of the **FillWithDataRowAdded** event will help us a lot to implement our task:

```
private void iGrid1_FillWithDataRowAdded(
  object sender, iGFillWithDataRowAddedEventArgs e)
{
  iGrid1.Rows[e.RowIndex].Tag = e.DataRow["Country"];
}
```

To access the corresponding row of the data source in the event handler, use either the **DataRow** or **DataReader** property of the event arguments object (**iGFillWithDataRowAddedEventArgs**). **DataRow** is used if you specified a **DataTable** or **DataView** as the data source in the **FillWithData** method, **DataReader** is used if the data source is an **IDataReader** or **IDbCommand** object. In the latter case the **DataReader** object is a whole DataReader used to read the data source. It is positioned on the row used to populate the corresponding grid row, and you can use all its methods including those that give you the best performance (the type-specific methods like **OleDbDataReader.GetInt32()**). However, don't do any other things which could close the **DataReader** or change its current row as it will affect the proper functionality of the **FillWithData** method.

28.1.4. Populating Empty iGrid with Only Data Columns

In some situations you may want to populate an empty iGrid with the columns from a data source and nothing else. To implement this task, you can call one of the **FillWithData** overloads that has the **dataStartRow** and **dataRowCount** parameters — just specify 0 for both parameters in such a call like in the following example:

```
iGrid1.FillWithData(dataSource, 0, 0);
```

iGrid also provides a method to implement this task more elegantly — **FillWithDataCols**. It has just one parameter, **dataSource**, used as the source of columns to create in iGrid. The previous call could be rewritten like the following one with the help of **FillWithDataCols**:

```
iGrid1.FillWithDataCols(dataSource);
```

There is one significant difference compared to the **FillWithData** call above: if your data source is a data reader object (an object implementing the **IDataReader** interface), it is closed after calling **FillWithData**. This does not happen in the case of the **FillWithDataCols** method, which allows you to continue working with the data reader without reopening it again.

The **FillWithDataCols** method is very helpful when you populate iGrid with data and its columns will contain combo box cells linked to drop-down lists. To link cell values from such a column to the

corresponding items in the drop-down list, iGrid must know which drop-down list is used in every column before you call the **FillWithData** method that adds data rows. In other words, first you need to create all grid columns corresponding to the data source, then set their **CellStyle**.**DropDownControl** property, and only after that call **FillWithData**. The **FillWithDataCols** method allows you to implement the first step as one simple statement. The whole scenario could look like this:

```
iGrid1.FillWithDataCols(tbl);
iGrid1.Cols["Type"].CellStyle.DropDownControl = dropDownListCustomerType;
iGrid1.FillWithData(tbl, true);
```

Pay attention to the last statement. We call the overloaded version of **FillWithData** in which the **useCurColSet** is set to True. If we did not specify this option, we would lose the columns created above.

28.2. Populating iGrid Drop-Down Lists with Data

You can use the **FillWithData** method of iGrid to fill it with data from an ADO.NET data source. iGrid built-in drop-down lists implemented in the **iGDropDownList** class also provide you with a similar method. The **FillWithData** method of the **iGDropDownList** class can be used to populate a drop-down list with the data from a data source. Such drop-down lists are often used in relational databases to provide lists of values for combo box cells. As a rule, the two tables are linked by a numeric field containing unique values (id, code, etc.)

The **FillWithData** method for drop-down lists has two overloads:

```
public void FillWithData(
  object dataSource, string itemValueCol);
public void FillWithData(
  object dataSource, string itemValueCol, string itemTextCol);
```

The first of them is used when list item texts and values equal; the second version is used when item texts and corresponding values differ.

Let's consider a grid displaying customer data as an example. Let's suppose we have the main table Customers with all these data. This table has a field with the name TypeID containing numeric values of customer type codes. Names of these types are stored in another table, CustomerTypes, with two fields — ID and Name. We want to show all these data in iGrid, and the column with customer types must allow the user to select the customer type from a drop-down lists. This task can be implemented with the following code:

```
dropDownListCustomerTypes.FillWithData(
  dataTableCustomerTypes, "ID", "Name");

gridCustomers.FillWithDataCols(dataTableCustomers);
gridCustomers.Cols["TypeID"].CellStyle.DropDownControl =
  dropDownListCustomerTypes;
gridCustomers.FillWithData(dataTableCustomers, true);

gridCustomers.Cols.AutoWidth();
```

The first statement populates an instance of **iGDropDownList** with the type codes and their names — the names of the corresponding fields are specified in the parameters of the **iGDropDownList**.**FillWithData** method. The next statement calls the **FillWithDataCols** method of iGrid to create grid columns corresponding to all columns in the Customers table. The key point in this call is that iGrid column keys will be set to the names of the fields in the Customers table. This fact is used in the next statement in which we specify the drop-down list for the column with

the key "TypeID". iGrid must know the drop-down list to link cell values to the corresponding items in the drop-down list before we call the **FillWithData** method that adds data rows. Note that the **useCurColSet** parameter (the second parameter of **FillWithData**) is set to True to use the column set we already defined.

The last statement in which we call the **AutoWidth** method of the **Cols** collection of iGrid is optional but can help the user a lot. This call automatically adjusts the width of every column so that cell values are displayed without clipping.

28.3. Data Binding Sample for DataTable

28.3.1. Copying iGrid Changes to DataTable

This subsection gives you an idea how you can bind iGrid to the data in an ADO.NET **DataTable** object and edit them. The first topic of this subsection demonstrates how you can copy the changes in iGrid into the underlying DataTable, which is enough in many cases. If the underlying DataTable object can be edited from another parts of your software, you also need to reflect these changes in iGrid. The next topic, Displaying DataTable Changes in iGrid, explains how to implement this.

To copy grid changes into the underlying data source, we need to link every grid row to the corresponding row in the DataTable. The **DataRow** object representing a DataTable row is a right choice for that. You can populate iGrid with data in one statement using its **FillWithData** method. After that, save DataRow objects for every row from the DataTable in the **Tag** property of iGrid rows. If a reference to iGrid is stored in the fGrid form field and the fDataTable form field contains a reference to the DataTable, the method that populates iGrid may look like this:

```
private void LoadData()
{
  // Load data into the grid.
  fGrid.FillWithData(fDataTable);

  // Save the links to DataRows in row tags.
  for(int iRow = 0; iRow < fGrid.Rows.Count; iRow++)
    fGrid.Rows[iRow].Tag = fDataTable.Rows[iRow];
}
```

Having this back link, it's easy to copy new cell values from iGrid into the DatTable after the user has edited a cell. Generally the **BeforeCommitEdit** event of iGrid is used for that. In the simplest case, the corresponding event handler will look like the following one:

```
private void fGrid_BeforeCommitEdit(
  object sender, iGBeforeCommitEditEventArgs e)
{
  fGridChange = true;

  DataRow myDataRow = (DataRow)fGrid.Rows[e.RowIndex].Tag;
  string myColumnName = fGrid.Cols[e.ColIndex].Key;
  myDataRow[myColumnName] = e.NewValue;

  fGridChange = false;
}
```

Notice the extra Boolean fGridChange flag we set before any changes in DataRow objects of the DataTable. The usage of this flag is explained in the next topic dedicated to processing DataTable events to reflect changes from the DataTable in iGrid. However, if you do not need that second part

of the data binding functionality, you can safely remove all statements with this flag from the code snippets.

Pay also attention to the fact that iGrid column keys correspond to DataTable column names because we populated iGrid with the **FillWithData** method that sets column keys automatically.

The code snippet above works well for fields with string data when there are no restrictions on the entered strings. However, for fields of other types (numeric, date fields and the like) the string entered by the user may contain illegal characters or may not be converted to the corresponding data type. One of the built-in features of iGrid that will help us to validate entered data on the iGrid side automatically is its ability to convert the entered value to the type of the edited cell value. If it cannot be done, iGrid informs you about that with the corresponding message box. You can find out more about this feature of iGrid from the Getting Cell Values from Entered Strings topic.

But such a general data validation is just a half of the validation process in real-world scenarios. The underlying DataTable may have its own field value constraints, such as foreign key constraint. If it is the case, an exception may occur when you save the new field value in the DataRow object. If you want to process this situation with a custom logic, you can also do this on the side of iGrid. One of the possible solution is to use the try-catch block to intercept exceptions thrown when you try to save a new value in the data row:

```
private void fGrid_BeforeCommitEdit(
  object sender, iGBeforeCommitEditEventArgs e)
{
  fGridChange = true;

  DataRow myDataRow = (DataRow)fGrid.Rows[e.RowIndex].Tag;
  string myColumnName = fGrid.Cols[e.ColIndex].Key;

  try
  {
    myDataRow[myColumnName] = e.NewValue;
  }
  catch (Exception ex)
  {
    MessageBox.Show(ex.Message, "Data validation error",
      MessageBoxButtons.OK, MessageBoxIcon.Error);
    e.Result = iGEditResult.Proceed;
  }

  fGridChange = false;
}
```

The two other main operations with a DataTable we have to implement in iGrid are row addition and row deletion. You can add a new row to both iGrid and the DataTable using a method like this:

```
private void AddRow()
{
  fGridChange = true;

  DataRow myDataRow = fDataTable.Rows.Add();
  iGRow myRow = fGrid.Rows.Add();
  myRow.Tag = myDataRow;
  fGrid.SetCurCell(fGrid.Rows.Count - 1, 0);

  fGridChange = false;
}
```

Row deletion can be done with the help of this method:

```
private void DeleteRow()
{
  if (fGrid.CurCell != null)
  {
    fGridChange = true;

    int myRowIndex = fGrid.CurCell.RowIndex;
    DataRow myDataRow = (DataRow)fGrid.Rows[myRowIndex].Tag;
    myDataRow.Delete();
    fGrid.Rows.RemoveAt(myRowIndex);

    fGridChange = false;
  }
}
```

These two row manipulation methods can be invoked when the user presses the INSERT and DELETE keys while iGrid has input focus:

```
private void fGrid_KeyDown(object sender, KeyEventArgs e)
{
  switch (e.KeyCode)
  {
    case Keys.Insert:
      AddRow();
      e.Handled = true;
      break;
    case Keys.Delete:
      DeleteRow();
      e.Handled = true;
      break;
  }
}
```

If the DataTable used to populate iGrid can be edited in another part of your software, you also need to reflect DataTable changes in iGrid. The next topic, Displaying DataTable Changes in iGrid, shows how you can implement this.

28.3.2. Displaying DataTable Changes in iGrid

In the case if you populated iGrid with the data from a **DataTable** object and you need to reflect changes in the underlying DataTable in iGrid, you can use the technique described below.

To implement what we need, we should process some DataTable events notifying us about corresponding changes. The arguments of these events contain a reference to the DataRow upon which an action has occurred. We can use the passed DataRow object to find the corresponding row in iGrid and perform the same action with it. The following helper method will be used for this purpose in our event handlers:

```
private iGRow GridRowFromDataRow(DataRow dataRow)
{
  foreach (iGRow row in fGrid.Rows)
  {
    if ((DataRow)row.Tag == dataRow)
      return row;
  }

  // The row must be found, but if something went wrong,
  // inform about that with an exception.
  throw new Exception("iGrid row not found");
}
```

Let's see how to display changes in the DataTable's field values in iGrid. The **DataTable** class provides us with the **ColumnChanging** event to notify about such changes:

```
private void fDataTable_ColumnChanging(
  object sender, DataColumnChangeEventArgs e)
{
  if (fGridChange)
    return;

  if (e.Row.RowState != DataRowState.Detached)
  {
    iGRow myGridRow = GridRowFromDataRow(e.Row);
    string myColKey = e.Column.ColumnName;
    myGridRow.Cells[myColKey].Value = e.ProposedValue;
  }
}
```

Remember the fGridChange flag mentioned in the preceding topic? If we copy changes in iGrid into the DataTable by changing its rows in our code, the DataTable events like **ColumnChanging** are triggered. If we do not ignore them in these situations, the actions from the event handlers will be executed too — which is obviously is not what we want to happen.

If a new row is added to the DataTable, its **RowChanged** event will help us to display the new row in iGrid:

```
private void fDataTable_RowChanged(
  object sender, DataRowChangeEventArgs e)
{
  if (fGridChange)
    return;

  if (e.Action == DataRowAction.Add)
  {
    iGRow myRow = fGrid.Rows.Add();
    myRow.Tag = e.Row;

    foreach (DataColumn col in fDataTable.Columns)
      myRow.Cells[col.ColumnName].Value = e.Row[col.ColumnName];

     fGrid.SetCurCell(fGrid.Rows.Count - 1, 0);
  }
}
```

Pay attention to the fact that some fields of the new DataRow may be initialized with default or auto-calculated values. We copy them to iGrid with the foreach loop above. The **SetCurCell** method is used to highlight the new row. When we call it, iGrid also automatically scrolls its contents to make the new row visible in the viewport.

The last possible operation with DataTable rows is row deletion. The DataTable notifies us about these operations with the **RowDeleting** event. We will use it to reflect the corresponding changes in iGrid:

```
private void fDataTable_RowDeleting(
  object sender, DataRowChangeEventArgs e)
{
  if (fGridChange)
    return;

  int myRowIndex = GridRowFromDataRow(e.Row).Index;
  fGrid.Rows.RemoveAt(myRowIndex);
}
```

28.3.3. Optimization for DataTable with Primary Key

The data binding functionality described in the two preceding topics is based on **DataRow** objects stored in iGrid row tags. When we need to find the iGrid row corresponding to a DataRow, we enumerate all iGrid rows from top to bottom until we find the given DataRow. This approach works fast enough even for data sets with a hundred thousand rows on modern computers (usually it takes 20-30 milliseconds). However, if you experience performance problems with this approach for some reasons, there is a way to speed up our functionality if the DataTable has a primary key. The idea is to use primary key values as iGrid row keys and employ the built-in fast algorithm to search iGrid rows by row keys.

Pay attention to the fact that this optimization may be required only if you reflect changes from the underlying DataTable into iGrid. If you copy changes only in one direction from iGrid to the DataTable, this part of the functionality does not require any optimization.

To make the new approach with primary keys work, we just need to make 4 small changes in the code snippets from the preceding two topics. But first let's introduce a new string constant containing the name of the primary key field to make the code easily reusable, for example:

```
private const string cPrimaryKey = "ID";
```

The first method we modify is the method used to upload data into iGrid. This time we will populate our grid with the special overloaded version of the **FillWithData** method that allows us to specify the row key column and display it in iGrid:

```
private void LoadData()
{
  // Load data into the grid.
  fGrid.FillWithData(fDataTable,
    false, cPrimaryKey, true, null, false, false);

  // Save the links to DataRows in row tags.
  for(int iRow = 0; iRow < fGrid.Rows.Count; iRow++)
    fGrid.Rows[iRow].Tag = fDataTable.Rows[iRow];
}
```

Now iGrid row keys store ID values of the corresponding DataTable rows and we can use them to find iGrid rows quickly due to the iGrid internal index. The updated version of the GridRowFromDataRow function will look even simpler now:

```
private iGRow GridRowFromDataRow(DataRow dataRow)
{
  string myRowKey = dataRow[cPrimaryKey].ToString();
  return fGrid.Rows[myRowKey];
}
```

Note that primary keys are not always strings, and we need to convert them to strings to use as iGrid row keys.

The new event handler of the **ColumnChanging** event of the **DataTable** class must also update the row key of the corresponding iGrid row if the value of the primary key has changed:

```
private void fDataTable_ColumnChanging(
  object sender, DataColumnChangeEventArgs e)
{
  if (fGridChange)
    return;

  if (e.Row.RowState != DataRowState.Detached)
  {
    iGRow myGridRow = GridRowFromDataRow(e.Row);
    string myColKey = e.Column.ColumnName;
    myGridRow.Cells[myColKey].Value = e.ProposedValue;

    // Update row key if required:
    if (myColKey == cPrimaryKey)
      myGridRow.Key = e.ProposedValue.ToString();
  }
}
```

And the last change must be done in the event handler of the **RowChanged** event notifying us about creation of new rows in the DataTable. We need to add just one statement that sets the row key of the created row:

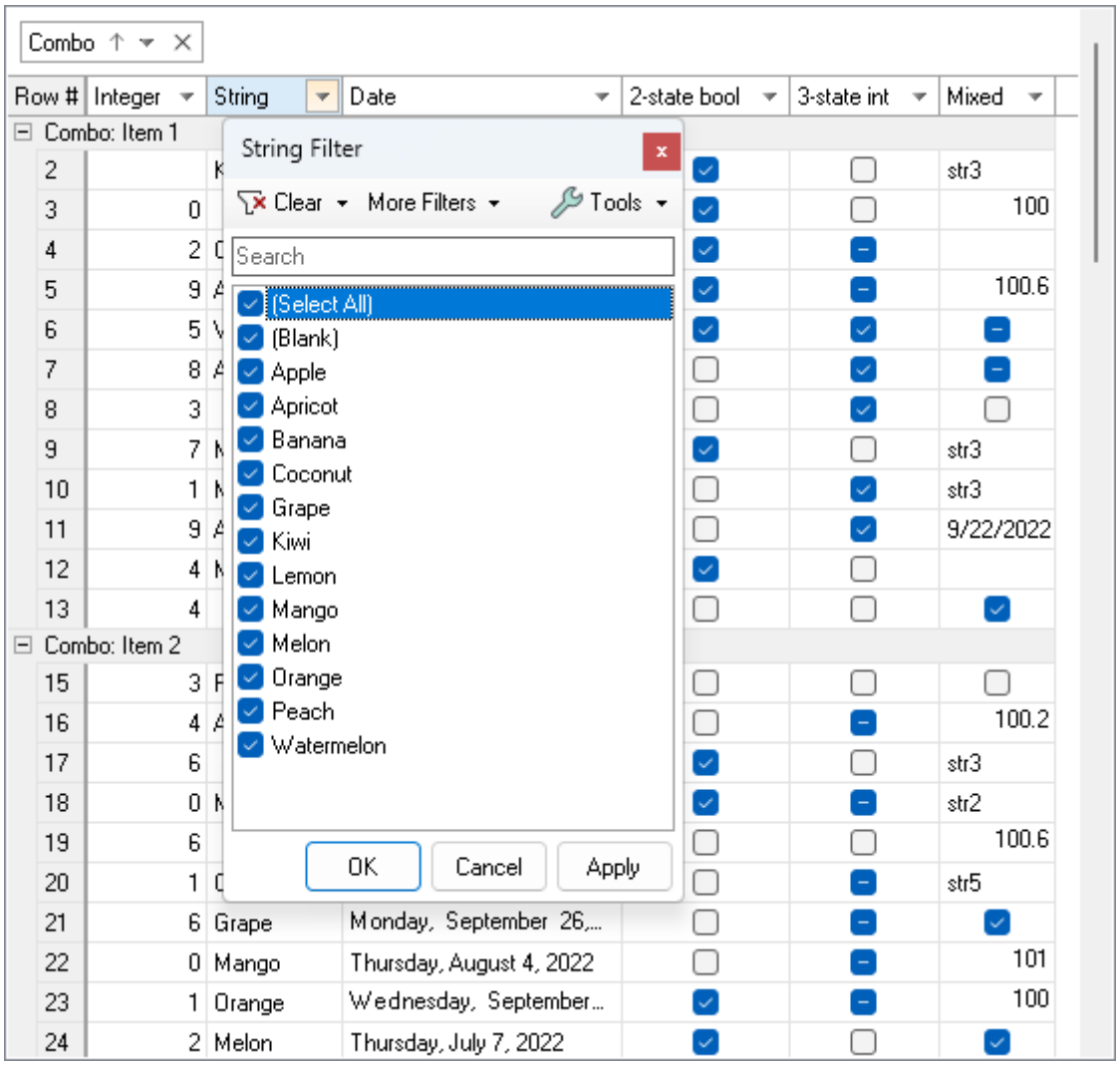
```
private void fDataTable_RowChanged(object sender, DataRowChangeEventArgs e)
{
  if (fGridChange)
    return;

  if (e.Action == DataRowAction.Add)
  {
    iGRow myRow = fGrid.Rows.Add();
    myRow.Tag = e.Row;

    // Set row key:
    myRow.Key = e.Row[cPrimaryKey].ToString();

    foreach (DataColumn col in fDataTable.Columns)
      myRow.Cells[col.ColumnName].Value = e.Row[col.ColumnName];

    fGrid.SetCurCell(fGrid.Rows.Count - 1, 0);
  }
}
```

29. FILTERING WITH THE AUTOFILTERMANAGER ADD-ON

29.1. Introduction to AutoFilterManager

AutoFilterManager is an add-on for iGrid.NET that adds the autofilter functionality to iGrid. This component is available for an extra fee for iGrid.NET users.

AutoFilterManager was designed to provide your users with a very handy tool to build common grid filters with minimal efforts. This concept affected the design and functionality of all parts of AutoFilterManager. For instance, the filter box dialog can be resized and even moved to see all grid cells. The dialog has a well thought-out keyboard interface one can use to do all grid filtering operations from the keyboard. Its custom filter system is very simple and intuitive, but versatile so that users can apply any condition to any type of data.



The AutoFilterManager functionality was modelled on the standard autofilter dialog in Microsoft Office applications, such as Excel and Access, to provide users with the familiar interface. But iGrid's autofilter does not have some disappointing restrictions of the Microsoft autofilter functionality. Among them the Apply button in the filter box, an unlimited number of custom filter rules, and the movable filter box dialog. Moreover, 10Tec AutoFilterManager also allows the users to save grid filter presets in memory or files and restore saved filter presets later.

One more key point of 10Tec AutoFilterManager is that it was optimized for real-world grids. It has excellent performance even on huge grids with 100,000+ rows and many columns.

AutoFilterManager's programming model is very simple. To add the autofilter functionality to a grid, you just drop the AutoFilterManager component onto the form and link it to the target grid with the **Grid** property. That's all — when you launch the application, your grid will show filter buttons in its column headers one can use to open the autofilter dialog to build filters. No coding.

The object design of AutoFilterManager provides you with objects to customize its user interface. First, all the colors and gradients of all constituent parts (filter box with its controls and menus, filter buttons in the target grid) can be adjusted to suit the color scheme of your application. Second, all the interface texts you can see in AutoFilterManager can be changed, which is generally used for software localization.

29.2. Main Features and Benefits

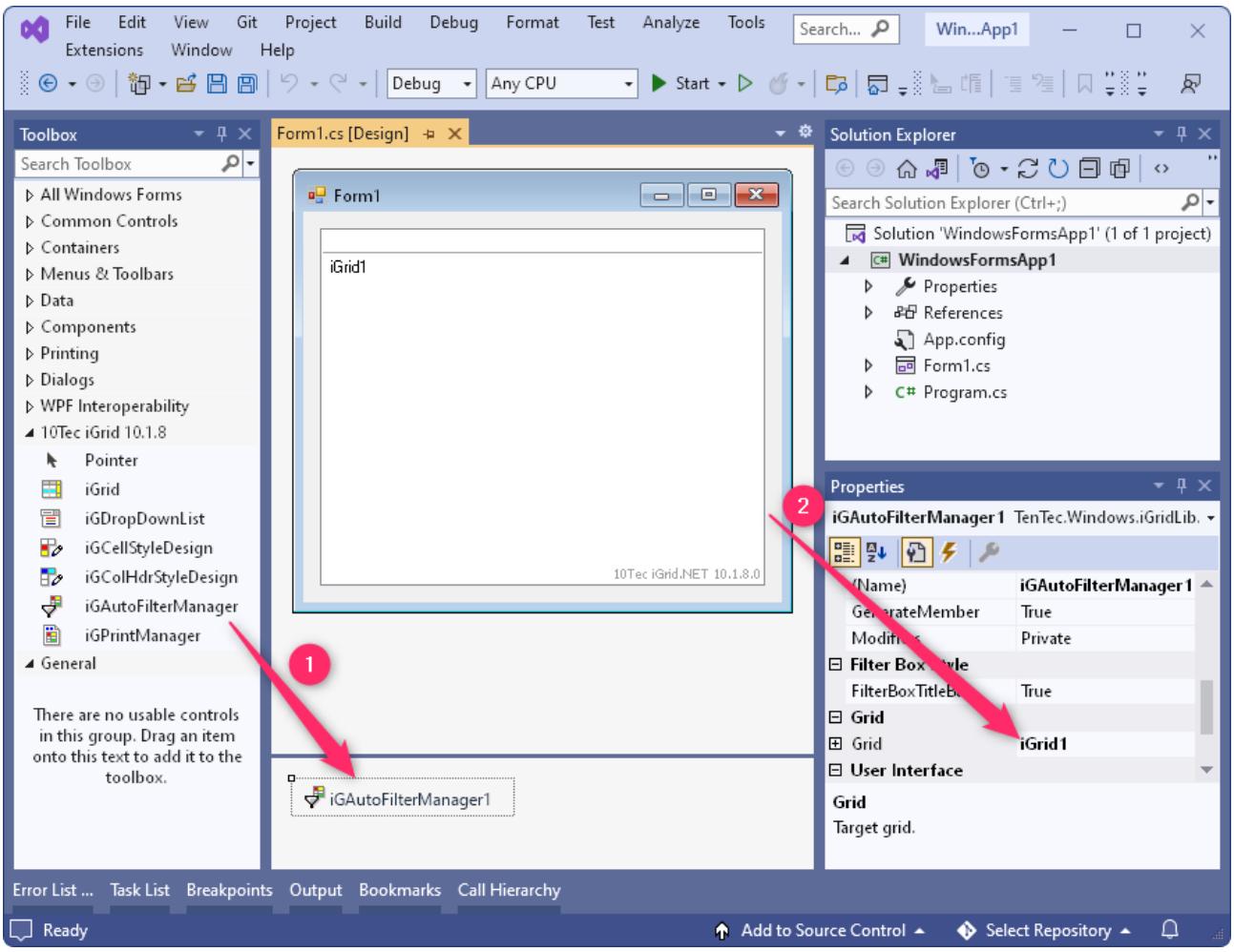
The iGrid.NET AutoFilterManager component allows you to perform the following tasks:

1. When AutoFilterManager is attached to an iGrid control, the target grid automatically displays drop-down filter buttons in its column headers. These buttons are used to open the autofilter dialog (also known as "filter box") to define various column filters.
2. The filter button indicates whether a filter is defined for a column with a special funnel icon displayed next to the down arrow glyph. The effective filter criteria can be viewed in the tooltip displayed for the filter button without opening the autofilter dialog.
3. The filter box is also used to perform such operations with the whole grid filter like saving to or restoring from a file, or clearing filters from all columns. No additional controls appear in the target grid for these functions not to overburden the user interface.
4. All operations in the filter box can be performed from the keyboard. The filter box itself can be opened using any key or key combination defined by the developer.
5. The filter box is resizable so you can grab and drag its edges to get the best size. It also has an optional title bar that can be used to drag the entire filter box to a new position to see all cell values in the filtered column.
6. The filter box has the Apply button one can use to adjust filter parameters and see the result of filtering in the target grid without closing and reopening the filter box over and over again.
7. AutoFilterManager works not only with all basic data types like numbers, dates and strings, but also can be used to process all other data types like enumeration values and even custom objects. All iGrid cell types — normal text cells, combo box cells and check box ones — are supported. AutoFilterManager can be even used for columns containing a mix of different cell types and cell values of different data types.
8. The check list of existing items to filter displays only the items that are visible in the grid when the filters in other columns are in effect. This can save a lot of time when working with huge grids as the user will see only with the reduced list of available items but not all possible items from the grid.
9. AutoFilterManager allows you not only to tick existing column items to filter by them, but also to define custom filter condition using major filter operators ("contains", "greater than", etc.). The custom filter can be used together with ticked items in contrast to the autofilter functionality in Microsoft Office, when you can use either the item list to filter or a custom filter criteria.
10. One of the limitations of the Microsoft Office autofilter is just 2 custom filter conditions. AutoFilterManager allows you to define an unlimited number of them!
11. AutoFilterManager has a simple but powerful set of custom filter operators. They are not specific to particular data types, and thus you can use string filters like "contains" to filter numeric or even date values. The custom operators are case-insensitive for more convenience.
12. The performance of AutoFilterManager is good on huge grids which have 100,000 or even more rows. Its filter box item list is never empty like it can be in Microsoft Access if the number of unique items in a column exceeds 1,000.

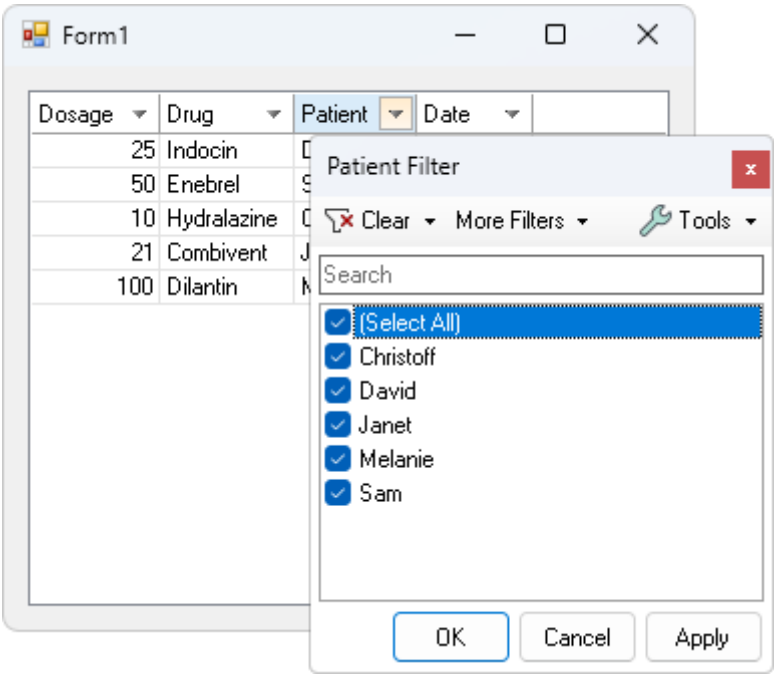
13. AutoFilterManager provide an option to specify whether to apply the current filter automatically as soon as any cell has been edited or wait until the moment when the user decides to reapply filter.
14. AutoFilterManager can be used to filter not only "flat" tables but also grids grouped by several columns.
15. All interface strings of AutoFilterManager are customizable and can be localized. The colors and fonts of the filter box and filter button in column headers of the target grid can be adjusted to meet the application color scheme and style.
16. The filter for one column or the whole grid can be serialized to an XML string that can be saved to and restored from any textual store (a file, registry, etc.).

29.3. Common Usage Scenario

To add the autofilter functionality to your grid, create an instance of the **iGAutoFilterManager** component and link it to the target grid. The easiest way to do that is to drop the **iGAutoFilterManager** component onto the form and select the target grid in its **Grid** property:



That's enough to enable the autofilter feature in the grid. Now, when you launch the application and add some data to the grid, AutoFilterManager automatically adds its filter buttons to the column headers one can use to define different filter criteria:



Here is the code behind the sample form:

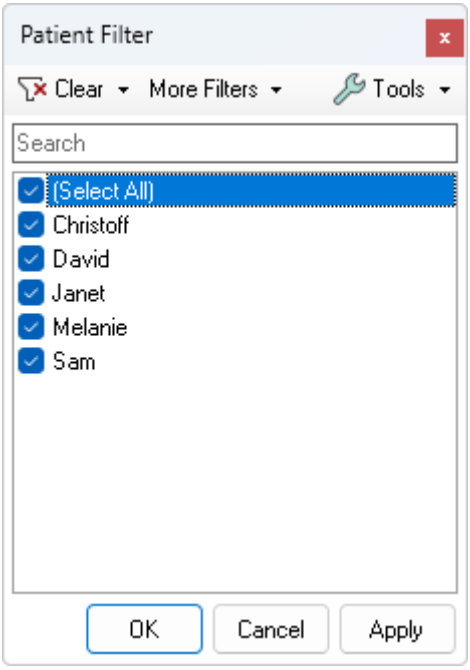
```
public partial class Form1 : Form
{
  public Form1()
  {
    InitializeComponent();
  }

  private void Form1_Load(object sender, EventArgs e)
  {
    iGrid1.FillWithData(GetTable());
    iGrid1.Cols["Date"].CellStyle.FormatString = "{0:d}";
    iGrid1.Cols.AutoWidth();
  }

  private DataTable GetTable()
  {
    DataTable table = new DataTable();

    table.Columns.Add("Dosage", typeof(int));
    table.Columns.Add("Drug", typeof(string));
    table.Columns.Add("Patient", typeof(string));
    table.Columns.Add("Date", typeof(DateTime));

    table.Rows.Add(25, "Indocin", "David", DateTime.Now);
    table.Rows.Add(50, "Enebrel", "Sam", DateTime.Now);
    table.Rows.Add(10, "Hydralazine", "Christoff", DateTime.Now);
    table.Rows.Add(21, "Combivent", "Janet", DateTime.Now);
    table.Rows.Add(100, "Dilantin", "Melanie", DateTime.Now);

    return table;
  }
}
```

Note that we have written no code to make autofiltering work.

29.4. Filter Box Features

29.4.1. Filter Box Overview

When you open the filter box for a column, by default it looks like this dialog:



Every AutoFilterManager filter box has the following main parts:

- **Menu bar**

  The filter box menu system contains different commands that are mainly applied to the filter for the current column or the whole grid (clearing the filter, saving/restoring it to or from a file, adding custom filter conditions, etc.). You can also find special commands which affect the filter box itself or other "action" commands in the menu.

- **Search box**

  This text field above the item list provides you with the ability to find the required items in the item list and is used to filter it by a substring. The filtering is performed as soon as you enter every new character.

  If you filtered the item list using this control, only the visible checked items will be included in the filter.

- **Filter item list**

  The central part of any filter box is a checklist of all unique cell texts you can see in the column. In most cases people filter a grid with this pick list: they simply check items they want to see in the grid. To quickly set or clear the check mark for all items, use the first special item called "Select All".

  AutoFilterManager's filter item list is sorted ascending helping you to find the items you need. You can also find the required item by first characters using incremental search: to do that, start typing when the filter item list has input focus.

- **OK, Cancel and Apply**

  The OK button allows you to apply the defined filter to the grid and close the filter box. The Cancel button is used to abandon all your changes in the filter box and close it. Using the Apply button, you can apply the defined filter to the grid without closing the filter box.

The filter box dialog is implemented like a normal window. By default it has a title bar, and the caption includes the column name the filter box is displayed for. The filter box dialog can be moved to any position on the screen by dragging its title bar. The size of the filter box can also be changed

by dragging its edges. AutoFilterManager remembers the last size of the filter box and restores it automatically when you display it the next time.

The filter box dialog is a non-modal window which is closed automatically when you click outside of it or bring another window to front. In this case any changes in the filter box aren't saved when closing.

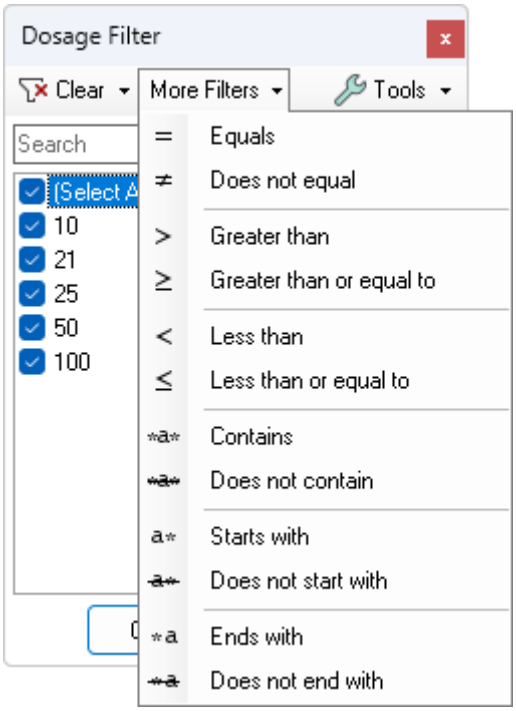
29.4.2. Custom Filter Section

Defining custom filter

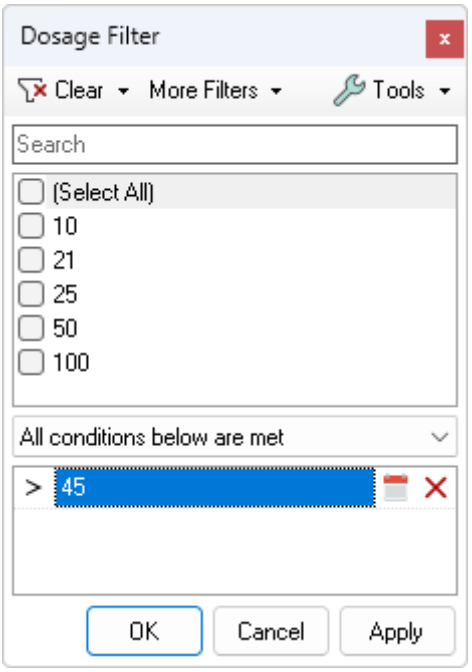
In addition to the ability to check items to display, AutoFilterManager allows you to define custom filters based on the following operators and their negative equivalents:

- "Equals"
- "Greater Than" and "Greater Than Or Equal To"
- "Contains"
- "Starts With" and "Ends With"

To define a new custom filter condition, first select the required operator from the "More Filters" drop-down menu in the filter box menu bar:



A new optional section called "Custom Filter" will be automatically displayed, and the selected operator will be added to it. To complete custom condition creation, enter the operator parameter like on the sample screenshot below:



The custom filter list allows you to add as many filters as you need. To add one more custom filter condition, select the required filter operator from the "More Filters" menu again.

If you have more than one custom condition, you need to tell AutoFilterManager whether they all should be true for an item to display it or it is enough to meet at least one custom condition. This is done with the combo box above the custom condition list. In fact, it specifies the AND or OR logical operator for the defined custom conditions.

You can also edit the custom conditions (change the operator and/or its parameter), or remove the unneeded conditions using the keyboard or mouse.

Custom conditions and the filter item list

Pay attention to the fact that the custom condition set is used **together** with the filter item list. This means that you can check several items in the item list and define additional custom criteria to filter the grid. In other words, the filter criteria made from the checked items and the custom condition set are joined together using the logical OR operator.

Note that AutoFilterManager automatically removes the check marks from all items if they all were selected when you apply a filter criteria with custom conditions. It would not make sense to have them all selected because all rows would be visible in this case even if custom conditions are defined.

Useful notes

1. All comparisons in custom conditions are case-insensitive.
2. All custom operators which usually used only for strings ("Contains", "Starts with", ...) can be used with numeric and even date values. In this case the values are simply treated like strings.
3. To enter a date as the parameter of a custom operator applied to date values, use any string representation available in your OS local settings. Generally these are long and short date formats. To avoid any ambiguity, you can also use the drop-down date picker opened when you click the calendar icon next to the parameter field. It enters the selected date in the local short date format.

29.4.3. Keyboard Control

The filter box controls and their tab order are optimized for fast work in most common scenarios. All filter box controls except its top menu bar can be accessed with the TAB key and the SHIFT+TAB combination, which traditionally move input focus to the next and previous control on a Windows

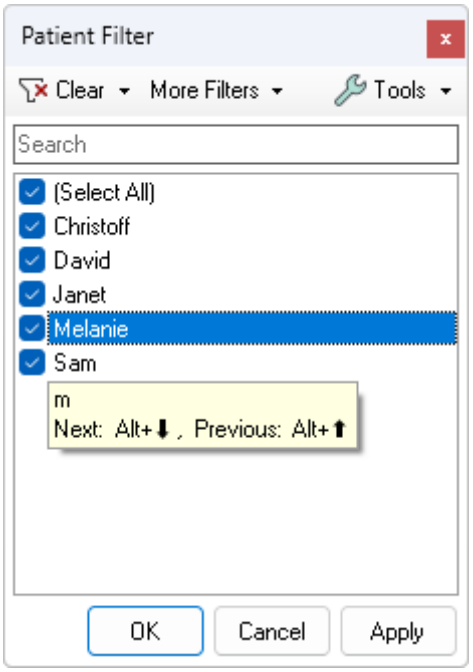
form. The items in the top menu bar can be accessed with the ALT+'letter' shortcuts like traditional Windows menu items.

Working with the filter item list

When you open the filter box, the item list always receives input focus. This allows you to quickly check/uncheck the items you want to see in the grid. To do that, you can use the UP ARROW and DOWN ARROW keys to move the selection to the required item and press SPACE to toggle the check mark. The HOME and END keys can be used to move the selection to the first or last item. Traditional list navigation commands like PAGE UP/PAGE DOWN are also supported.

The first special item "Select All" is always selected every time when the filter box is opened. It allows you to quickly check/uncheck all items using the same SPACE key.

The item list supports incremental search allowing you to find the required item by typing its first letters:



If incremental search is active, the BACKSPACE key removes the last entered character. You can move to the next/previous match using ALT+DOWN ARROW/UP ARROW. The SPACE key adds a space character to the search string during incremental search and cannot be used to toggle the check mark for the selected item — you should use SHIFT+SPACE for that. The ESC key cancels incremental search.

If you want to search a long item list using filtering by a substring, use the Search box above the list. If the item list is focused (the default state after opening the filter box), you can quickly move input focus to the search box by pressing SHIFT+TAB. You can also move input focus to the Search box by pressing the ALT+S keyboard combination, regardless of which control is currently selected. To clear all characters entered to the Search box, press ESC while this control has input focus.

Custom filter section

When the custom filter section is visible, you can move input focus to it from the keyboard by pressing the TAB key. The list of custom conditions is a grid with several columns representing condition operators, their parameters, the date picker button, and the icon used to remove a condition. You can use all cursor movement keys to select the cells of this grid.

When a cell is selected, use SPACE to activate it. Activation means different actions for different cells. For instance, if the column with condition operator icon is selected, SPACE opens the menu with all available custom operators so that you can replace the existing operator with a new one. If a cell with condition parameter is selected, SPACE activates text editing for it.

A selected cell with condition parameter can also be put into edit mode if you press F2 or start typing. To commit your changes, press ENTER; to discard your input, press ESC.

The combo box above the custom filter list specifying the AND or OR logical operator for the defined custom conditions can be selected with TAB/SHIFT+TAB as well. When it has input focus, you can use the traditional Windows command keys like F4 to open its drop-down list, to select the required item with the cursor movement keys and so on. AutoFilterManager also provides a very handy keyboard shortcut to toggle the selected item in this combo box without selecting it: ALT+E.

Activating top menu items

The items in the top menu bar can be activated using the ordinary way with the ALT key. When you press it, a letter in every menu item is underlined, which means you can use ALT+'letter' combination to active the corresponding menu (also known as "keyboard cues"). When a top menu is opened this way, its subitems also display underscored characters telling you that you can activate items by pressing the corresponding letters.

The OK, Cancel, and Apply buttons

The actions related to the OK and Cancel buttons are performed using the keys that are standard for all Windows dialogs. The OK is the default dialog button, and you can press ENTER instead of clicking this button. As expected, pressing ESC is equivalent to clicking Cancel. The action linked to the Apply button can also be executed with the ALT+A keyboard shortcut regardless of the selected control.

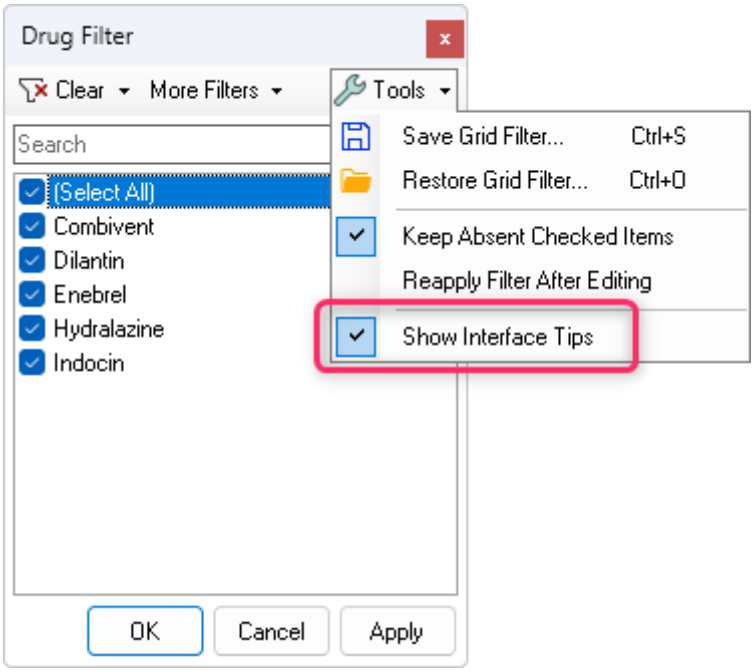
Note that ENTER and ESC do not work this way while you are editing the parameter of a custom filter condition because these keys are used to commit or discard condition parameter change.

The whole filter box dialog

If the AutoFilterManager filter box has a title bar (which can be programmatically turned on/off with the **FilterBoxTitleBar** property of the **iGAutoFilterManager** component), the ALT+SPACE combination opens the system window menu with the Move, Close and other related commands. You can use it to control the filter box window from the keyboard — like for any other normal Windows form.

29.4.4. Learning Mode

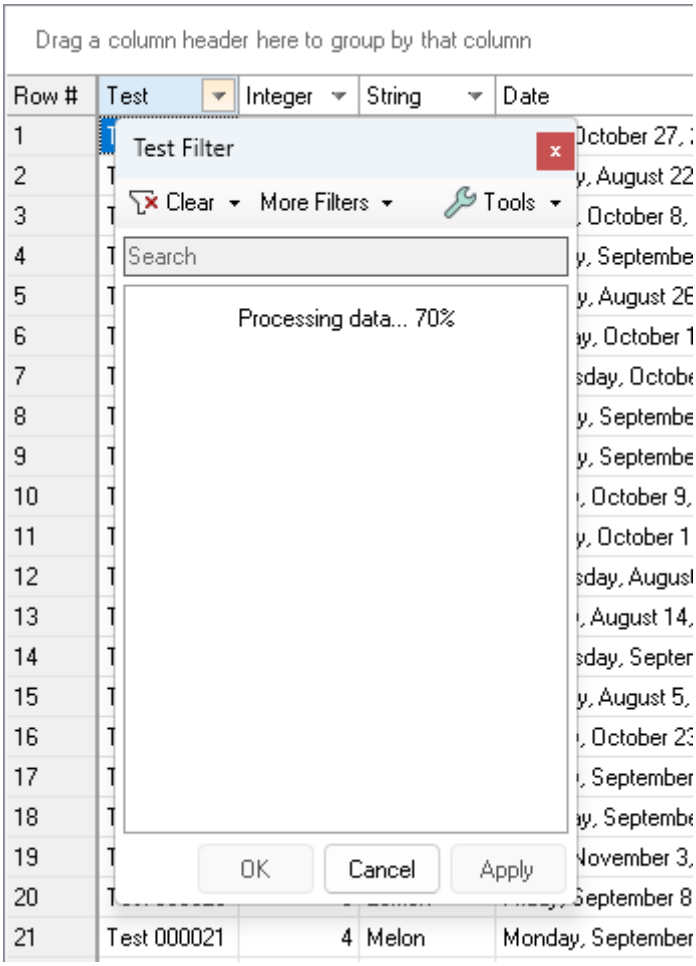
AutoFilterManager has a kind of learning mode when you can see a quick description for controls and the corresponding keyboard combinations in control tooltips. This mode is disabled by default. To activate it, check the "Show Interface Tips" item in the Tools menu:



29.4.5. Filtering Huge Grids

When the user opens the filter box, AutoFilterManager does a great deal of work to build the list of unique filter items that correspond to the current situation. The component scans all grid rows taking into account row visibility because rows may be already filtered out by filters in other columns, the scanned cell texts are analyzed for uniqueness to provide the list of unique items, these items are combined with the existing list of already checked items (AutoFilterManager can still show checked items with the addition "absent" if they are no longer present in the grid), and the resulting list of items is finally sorted. This complex process may take a significant amount of time and lead to non-responsive UI if it is done in the main UI thread when the filter box is opened. To prevent UI from freezing, AutoFilterManager does all this work in a separate background thread if you use the component with the default settings.

By default, AutoFilterManager waits half a second for this job to complete. If it was not enough, the filter box is opened anyway and becomes available for user actions. The list building process is finishing in the background, and the corresponding message informing about the progress of the process every 5% appears inside the filter item list:



This maximum display delay of the filter box can be adjusted with the **FilterBoxMaxDisplayDelay** property of the **iGAutoFilterManager** component. It is an integer property that specifies a value in milliseconds. The default value is set to 500 milliseconds.

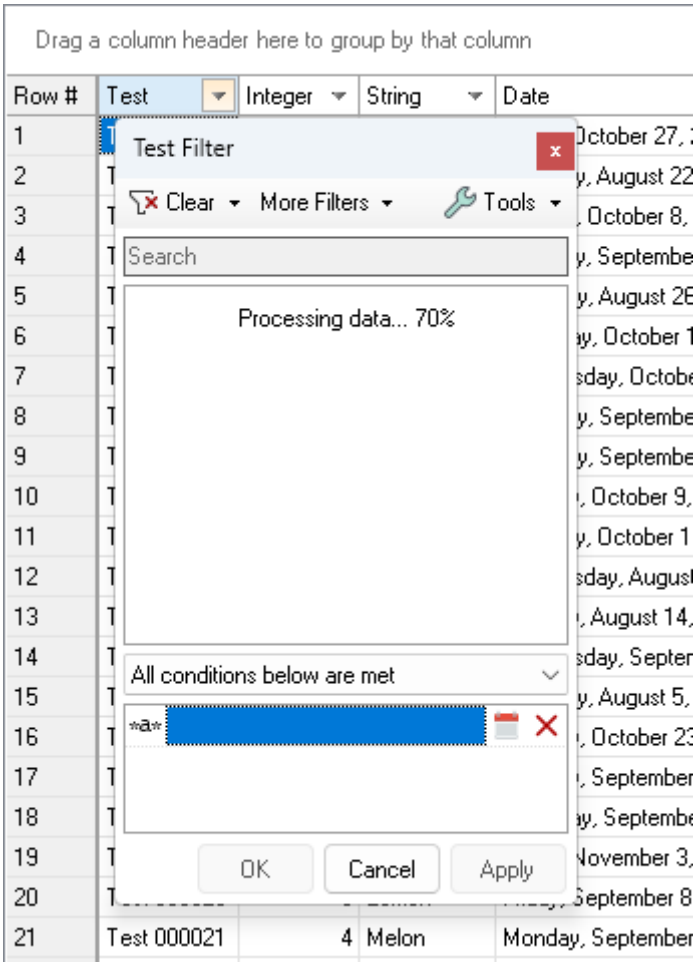
The template of the message displaying the list building progress can be changed/localized with the string property **iGAutoFilterManager**.**UIStrings**.**ItemList**.**BuildingProgressTemplate**. Its default value is set to "Processing data... {0}%'. The "{0}" substring in this string is replaced with the percentage of completion when displayed on the screen.

The **iGAutoFilterManager** class implements the **CustomConditionSuggestion** property of the **Nullable**<**iGFilterConditionOperator**> type. If this property is set to a value from the **iGFilterConditionOperator** enumeration, AutoFilterManager automatically creates a new custom filter condition with the specified operator in the filter box's list of custom filter conditions if this list was empty. The default value of this property is null (Nothing in VB), which means custom filter conditions are not created automatically the way described above.

Setting the **CustomConditionSuggestion** property to a non-null value may help your users to start using the filter box immediately after opening while the grid data is being processed to build the filter item list. For example, the following setting

```
iGAutoFilterManager1.CustomConditionSuggestion =
  iGFilterConditionOperator.Contains;
```

will result in the following state of the filter box while its filter item list is being populated:



Note that the user can already enter a value for the parameter of the Contains operator and click the OK or Apply button to apply the filter — even if the filter item list is not populated by that time yet.

if you want to prohibit building the list of filter items in the background thread for some reason, you can do this with the Boolean **BuildItemListInBackground** property of AutoFilterManager. To do this, set this property to False.

29.5. Basic Coding Techniques

29.5.1. Main Classes and Their Members

The iGAutoFilterManager class

When you drop the AutoFilterManager add-on on a form, an instance of the **iGAutoFilterManager** class is created. This is the most important class in the AutoFilterManager add-on. Its methods and properties are used to operate with the whole grid filter, and also to control the general aspects of look and behavior. For example, you can clear the filter from all columns in the grid using the **ClearFilter** method of this object:

```
iGAutoFilterManager1.ClearFilter();
```

The main method groups implemented by the **iGAutoFilterManager** class are as follows:

- Applying/reapplying the current filter: **ReapplyFilter**, **ReapplyFilterToRow**.
- Saving/restoring the grid filter: **SaveFilterToFile**/**RestoreFilterFromFile**, **SaveFilterToMemory**/**RestoreFilterFromMemory**.
- Clearing the grid filter: **ClearFilter**.

- Accessing the autofilter functionality for a particular column: **ColAutoFilter**.

All events related to filtering are also the events of the **iGAutoFilterManager** class. The main of them is the **FilterApplied** event. The properties of its event arguments object provide you with information how many rows were processed and filtered. You can also retrieve these values after applying a filter with the **ProcessedRowCount** and **FilteredRowCount** properties of the **iGAutoFilterManager** object.

Another useful event of the **iGAutoFilterManager** class is **CreateColAutoFilter**. It is raised before AutoFilterManager adds the filter button to a column header and can be used to prevent showing filter buttons in particular column headers.

The **FilterBoxOpened** and **FilterBoxClosed** events of the **iGAutoFilterManager** class inform you about the moments when a filter box becomes visible or hidden respectively.

The iGColAutoFilter class

All operations with the autofilter for one particular column are performed with the so-called column autofilter object. It is an instance of the **iGColAutoFilter** class retrieved with the **ColAutoFilter** method of the **iGAutoFilterManager** object. For example, you can use the **ShowFilterBox** method of an **iGColAutoFilter** object related to a column to display the filter box for the column from code:

```
iGAutoFilterManager1.ColAutoFilter(myColIndex).ShowFilterBox();
```

The **iGColAutoFilter** class redefines the standard **ToString** method so that it returns the text representation of a column filter for a human — what is displayed in the tooltip for the corresponding drop-down autofilter button.

iGFilterCriteria and related classes

Instances of the **iGFilterCriteria** class are used to construct filter criteria from code. They are created and populated by the developer solely from code and are passed to the column autofilter object when calling **iGColAutoFilter**'s **SetFilterCriteria** method.

Two other main classes whose instances are created implicitly or explicitly while defining a filter criteria from code are **iGFilterItem** and **iGFilterCondition**. They represent selected items and custom conditions in the filter box.

The functionality and usage examples of the **iGFilterCriteria** class and related types are described in greater detail in the Constructing Filter Criteria from Code section.

29.5.2. Saving/Restoring Grid Filter

The user can store the current filter for the whole grid in a file on a hard drive with the Save Grid Filter command from the Tools menu of the filter box. The same action can be performed from code using the **iGAutoFilterManager**.**SaveFilterToFile** method. The Restore Grid Filter from the filter box's Tools menu allows the user to load and apply the saved grid filter from a file. This operation can be performed from code with the **iGAutoFilterManager**.**RestoreFilterFromFile** method.

AutoFilterManager provides you with the ability to store grid filter as a named preset in memory. This feature is used when the user needs to switch between several temporary filters but does not want to save them on the hard drive as files. This functionality is implemented with the **SaveFilterToMemory**/**RestoreFilterFromMemory** methods of the **iGAutoFilterManager** class.

If you need to implement your own procedure that stores the grid filter to another source (text file, XML file, registry, etc.), you can use the **FilterAsXmlString** property of the **iGColAutoFilter** object that gives you a special string representation of the column filter.

29.5.3. Customizing AutoFilterManager

The behavior of AutoFilterManager and look of its visible parts can be adjusted mainly from code. This topic lists the available options. Unless otherwise noted, all properties mentioned in this topic are properties of the **iGAutoFilterManager** class.

Filtering behavior

The behavior of filtering algorithm is controlled with the two Boolean properties with self-descriptive names: **ReapplyFilterAfterEditing** and **ProcessFrozenRows**. The former option can also be changed by the user through the Tools menu of the filter box, the latter option can be set only by the developer.

Filter buttons in the target grid

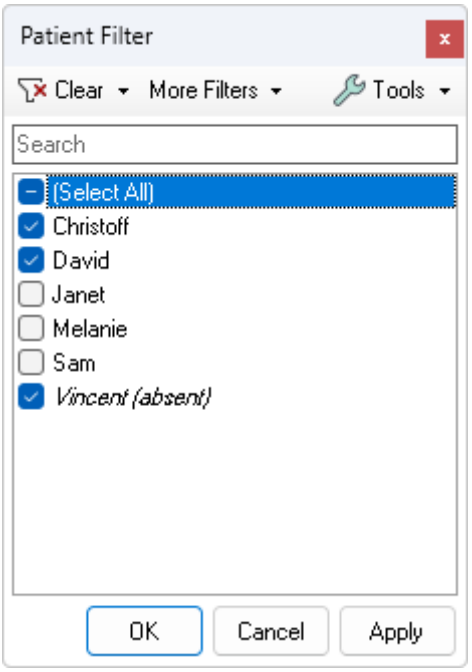
When you attach AutoFilterManager to a grid, it creates filter buttons in all columns by default. You may have columns that do not require filtering by some reasons. To avoid adding filter buttons to those columns, use the **CreateColAutoFilter** event raised before AutoFilterManager creates the filter button for a column in the target grid. The Boolean **DoDefault** property of the event argument object can be used to prevent AutoFilterManager from creating filter buttons for specified columns. For example, the following event handler can be used to show filter buttons in all columns except the first one:

```
private void fAutoFilterManager_CreateColAutoFilter(
  object sender, iGCreateColAutoFilterEventArgs e)
{
  if (e.ColIndex == 0)
    e.DoDefault = false;
}
```

The look (colors) of the filter button is adjusted with the **FilterButtonColors** object property of the **iGAutoFilterManager** object.

Filter box options

To control whether the filter box item list displays items that were checked previously for filtering but they are absent now in the grid, use the **KeepAbsentCheckedItems** property (also available for interactive change through the Tools menu). This option is very useful when you repopulate the grid with different data sets, and you need to save the user filter between data set change when a next set may not contain the checked items. If this option is off, the checked items are automatically removed from the item list if they are no longer present in the target grid. If this mode is on, the checked items are not removed but displayed in italic with the extra word "(absent)":



The **ShowInterfaceTips** property allows you to activate the AutoFilterManager learning mode in which you see short descriptions and the corresponding keyboard combinations in the tooltips for filter box controls. This option can also be toggled interactively in the Tools menu.

To adjust the colors and font of the filter box, use the subproperties of the **FilterBoxColors** and **FilterBoxFont** object properties of **iGAutoFilterManager**.

To save some space on the screen, you can remove the filter box title bar using the Boolean **FilterBoxTitleBar** property. The template of the filter box caption can be changed with the **iGAutoFilterManager**.**UIStrings**.**MainControlsTexts**.**FilterBoxCaptionTemplate** property (see notes regarding interface text customization below). If needed, you can completely ignore the suggested filter box caption and specify your own one dynamically with the help of the **FilterBoxDynamicCaption** event.

Interface texts customization

You can change or localize every string that appears in the AutoFilterManager UI. The **UIStrings** property is designated for this purpose.

It consists of several object subproperties. Each of them, in its turn, has lots of string properties that store all texts you can see in the interface of AutoFilterManager. These objects group interface strings into categories for more convenient use, some examples are **MainControlsTexts**, **CustomFilterOperators** and **Dialogs**.

When you change a property of the **UIStrings** object, AutoFilterManager automatically adjusts the layout of the filter box to fully display specified texts. If this adjustment occurs after every property assignment, the performance of your code may degrade significantly. To avoid this, disable the updates with the **UIStrings**.**BeginUpdate** method before a bulk change and enable them after finishing with the **UIStrings**.**EndUpdate** method.

29.6. Constructing Filter Criteria from Code

29.6.1. Types for Constructing Filter Criteria

AutoFilterManager provides you with the ability to create arbitrary filter criteria from code. The **iGFilterCriteria** class from the **TenTec.Windows.iGridLib.Filtering** namespace plays a central role in this process. Instances of this class are used to specify filter criteria for grid columns.

When the user defines a filter criteria in the AutoFilterManager filter box, they operate with the 2 filter box sections — item list and custom conditions. The user may select some items to filter by from the item list and/or may create some custom conditions like "Greater Than" or "Contains". The 5 main properties of the **iGFilterCriteria** class are designed to implement these tasks from code. The properties whose names start with "Select" are related to the item list, the other properties with the names starting with "CustomConditions" are related to the custom filter section:

PROPERTY	TYPE	DESCRIPTION
SelectAllItems	**Boolean**	Indicates whether all filter items are selected.
SelectedValues	**List**<**iGFilterItem**>	A list of values to filter by (for text cells).
SelectedCheckStates	**List**<**CheckState**>	A list of check box states to filter by (for check box cells).
CustomConditions	**List**<**iGFilterCondition**>	A list of conditions in the custom filter section.
CustomConditionsCombineMode	**iGFilterConditionsCombineMode**	The logical operator to combine custom filter conditions (And/Or).

The filter box item list can contain values of text cells and special items representing check states in check box cells to filter by. The **SelectedValues** and **SelectedCheckStates** properties of **iGFilterCriteria** are used separately for these 2 kinds of cells respectively. These properties are traditional .NET generic List objects containing elements of the data types specified in the table above. The **SelectAllItems** property indicates the check state of the special first item "(Select All)" in the filter box item list.

The **iGFilterItem** class was introduced to specify item list values to filter by. Its definition is placed below:

```
public class iGFilterItem
{
  public object Value;
  public string Text;

  public iGFilterItem(object value, string text)
  {
    Value = value;
    Text = text;
  }
}
```

An **iGFilterItem** object consists of two properties — **Value** of the **Object** type and **Text** of the **String** type. The former property, **Value**, is used to specify a value to filter by in its native format. The latter property, **Text**, is used to specify the corresponding string representation of the value on the screen. You must assign values to both properties because in the general case the value to filter by can differ from its string representation for the user (for example, if cell values are formatted using a format string).

The **CheckState** type in the table above is an enumeration from the **System.Windows.Forms** namespace in .NET Framework. The **iGFilterConditionsCombineMode** type is an AutoFilterManager enumeration containing 2 elements — **And** and **Or**.

The **iGFilterCondition** class was introduced to specify custom filter conditions. It is defined as follows:

```
public class iGFilterCondition
{
  public iGFilterConditionOperator Operator;
  public string Parameter;
}
```

The **iGFilterConditionOperator** type above is an enumeration containing all possible custom filter operators:

```
public enum iGFilterConditionOperator
{
  Equals,
  DoesNotEqual,
  GreaterThan,
  GreaterThanOrEqualTo,
  LessThan,
  LessThanOrEqualTo,
  Contains,
  DoesNotContain,
  StartsWith,
  DoesNotStartWith,
  EndsWith,
  DoesNotEndWith
}
```

Read the next two topics, Setting Filter Criteria and Reading Filter Criteria, to know how to use these types to create and read filter criteria from code.

29.6.2. Setting Filter Criteria

The **iGColAutoFilter** class representing the autofilter functionality for an iGrid column provides the **SetFilterCriteria** method to set filter criteria from code:

```
public void SetFilterCriteria(iGFilterCriteria filterCriteria);
```

Below you will find examples demonstrating how to use this method together with **iGFilterCriteria** objects to construct various filter criteria and apply them to iGrid from code. The filtering strategy is slightly different for different cell types. We will start from traditional text cells, then will show how to filter combo box cells, and finally will consider filtering check box cells.

The first example demonstrates how to define and apply a new filter criteria to filter a column containing integer values. Let's suppose we need to leave visible only the rows in which cell values equal 3. The following code implements this task:

```
iGFilterCriteria fc = new iGFilterCriteria();
fc.SelectedValues.Add(new iGFilterItem(3, "3"));
iGAutoFilterManager1.ColAutoFilter(0).SetFilterCriteria(fc);
iGAutoFilterManager1.ReapplyFilter();
```

If our cells were formatted as currency values and we lived in the US, we would need code like this:

```
iGFilterCriteria fc = new iGFilterCriteria();
fc.SelectedValues.Add(new iGFilterItem(3, "$3.00"));
iGAutoFilterManager1.ColAutoFilter(0).SetFilterCriteria(fc);
iGAutoFilterManager1.ReapplyFilter();
```

It can be tedious to specify formatted equivalents for values to filter manually. Moreover, if we have an international application that can work with different currency formats, we must have a way to specify the format string applied to every filter value to get its text representation. The **iGFilterCriteria** class provides 3 helper members, the **AddSelectedValue** method, the **AddSelectedValueFormatString** and **AddSelectedValueFormatProvider** properties, to automate this work.

The **AddSelectedValue** method has the following 2 overloaded versions:

```
public void AddSelectedValue(object value);
public void AddSelectedValue(object value, string text)
```

The second overloaded version with two parameters is not of much interest. It just creates an instance of the **iGFilterItem** class with the specified value and text, and then adds it to the **SelectedValues** list of the corresponding **iGFilterCriteria** object. What can help us a lot when we work with formatted cell values is the first overloaded version in conjunction with the **AddSelectedValueFormatString** and **AddSelectedValueFormatProvider** properties.

If you do not specify a format string in the **AddSelectedValueFormatString** property, the method adds the specified filter value with its text representation retrieved by calling the universal **ToString** method for the specified value. But if **AddSelectedValueFormatString** property contains a format string, this format string is applied to the specified value to get its text representation. This works exactly like format strings for grid cells — the text representation of the value is the result of the call **String.Format(AddSelectedValueFormatString, value)** (see the Cell Format Strings topic for more information). Gathering all these tools together, we could write the following code to specify a filter criteria to filter rows with sums 100, 150, and 200:

```
fc.AddSelectedValueFormatString = "{0:c}";
fc.AddSelectedValue(100);
fc.AddSelectedValue(150);
fc.AddSelectedValue(200);
```

In many cases the required format string is already specified in the cell style object for the corresponding column (the **CellStyle** property of the **iGCol** class). If so, an even better version of the code above could look as follows:

```
fc.AddSelectedValueFormatString =
iGrid1.Cols["sum"].CellStyle.FormatString;
fc.AddSelectedValue(100);
fc.AddSelectedValue(150);
fc.AddSelectedValue(200);
```

You can also use custom format providers — exactly like it can be done for iGrid cells with their **FormatString** and **FormatProvider** properties. If the format string specified in the **AddSelectedValueFormatString** property refers to a custom format provider, the implementation of this custom format provider (a class implementing the .NET **IFormatProvider** interface) should be assigned to the **AddSelectedValueFormatProvider** property, for example:

```
fc.AddSelectedValueFormatString = "{0:ip}";
fc.AddSelectedValueFormatProvider = new IPFormatProvider();
```

In this case the **AddSelectedValue** method will retrieve the text representation of the added value as the result of the call **String.Format(AddSelectedValueFormatProvider, AddSelectedValueFormatString, value)** — exactly as if you worked with grid cells formatted with a format provider (see the Cell Format Providers topic for more information).

Below is the next example demonstrating how to filter a column with string values. An iGrid containing fruit names in the column with the key "fruit" can be filtered to show only rows with apples and mangos using the following code:

```
iGFilterCriteria fc = new iGFilterCriteria();
fc.SelectedValues.Add(new iGFilterItem("Apple", "Apple"));
fc.SelectedValues.Add(new iGFilterItem("Mango", "Mango"));
iGAutoFilterManager1.ColAutoFilter("fruit").SetFilterCriteria(fc);
iGAutoFilterManager1.ReapplyFilter();
```

Below is a shorter equivalent based on the **AddSelectedValue** method call:

```
iGFilterCriteria fc = new iGFilterCriteria();
fc.AddSelectedValue("Apple", "Apple");
fc.AddSelectedValue("Mango", "Mango");
iGAutoFilterManager1.ColAutoFilter("fruit").SetFilterCriteria(fc);
iGAutoFilterManager1.ReapplyFilter();
```

In the case of string values the text representation of a value equals the value itself. As such, we can omit specifying text representation of the fruits and call the **AddSelectedValue** method with one parameter to write even shorter code:

```
iGFilterCriteria fc = new iGFilterCriteria();
fc.AddSelectedValue("Apple");
fc.AddSelectedValue("Mango");
iGAutoFilterManager1.ColAutoFilter("fruit").SetFilterCriteria(fc);
iGAutoFilterManager1.ReapplyFilter();
```

Pay attention to the call of the **ReapplyFilter** method of the **iGAutoFilterManager** object in the examples above. It performs actual filtering. The fact is that when you call the **iGColAutoFilter**.**SetFilterCriteria** method, the specified filter criteria is only loaded into the filter box and the corresponding internal structures, but it is not applied immediately. This allows you to define several filter criteria for several columns and apply the defined filter only once, avoiding unneeded filtering operation after every call of **iGColAutoFilter**.**SetFilterCriteria**. For the sake of demonstration of this ability, let's apply the previous two filters we considered above the most effective way:

```
iGFilterCriteria fc1 = new iGFilterCriteria();
fc1.AddSelectedValueFormatString = "{0:c}";
fc1.AddSelectedValue(100);
fc1.AddSelectedValue(150);
fc1.AddSelectedValue(200);
iGAutoFilterManager1.ColAutoFilter("sum").SetFilterCriteria(fc1);

iGFilterCriteria fc2 = new iGFilterCriteria();
fc2.AddSelectedValue("Apple");
fc2.AddSelectedValue("Mango");
iGAutoFilterManager1.ColAutoFilter("fruit").SetFilterCriteria(fc2);

iGAutoFilterManager1.ReapplyFilter();
```

Now let's demonstrate how to use the custom filter section from code. If we needed to filter the column with fruit names and display fruits with names starting with "O", we could write the following code:

```
iGFilterCriteria fc = new iGFilterCriteria();
fc.CustomConditions.Add(
  new iGFilterCondition(iGFilterConditionOperator.StartsWith, "O"));
iGAutoFilterManager1.ColAutoFilter("fruit").SetFilterCriteria(fc);
iGAutoFilterManager1.ReapplyFilter();
```

Like in the case of the **SelectedValues** list and the accompanying helper method **AddSelectedValue**, the **iGFilterCriteria** class provides a similar method to add custom conditions:

```
public void AddCustomCondition(
  iGFilterConditionOperator operator, string parameter)
```

The **AddCustomCondition** method can make our code shorter and cleaner:

```
iGFilterCriteria fc = new iGFilterCriteria();
fc.AddCustomCondition(iGFilterConditionOperator.StartsWith, "O");
iGAutoFilterManager1.ColAutoFilter("fruit").SetFilterCriteria(fc);
iGAutoFilterManager1.ReapplyFilter();
```

Sure, custom conditions can work together with selected values, for example:

```
iGFilterCriteria fc = new iGFilterCriteria();
fc.AddSelectedValue("Apple");
fc.AddSelectedValue("Mango");
fc.AddCustomCondition(iGFilterConditionOperator.StartsWith, "O");
iGAutoFilterManager1.ColAutoFilter("fruit").SetFilterCriteria(fc);
iGAutoFilterManager1.ReapplyFilter();
```

If you have empty cells, AutoFilterManager adds the special item "(Blank)" to the item list in the filter box to filter such cells. To do this from code, add an empty string to the **SelectedValues** list:

```
fc.AddSelectedValue(String.Empty);
```

A column with combo box cells is filtered using the same **SelectedValues** list. Combo box cells are actually text cells with attached drop-down lists, and you should specify the cell texts of corresponding drop-down list items to filter by in the **SelectedValues** list. If you know the item text, you can simply specify it as string like this:

```
fc.AddSelectedValue("Item text");
```

If you want to retrieve the item text to filter by from the corresponding drop-down list item, you can use the following universal approach:

```
fc.AddSelectedValue(iGDropDownList1.Items[1].ToString());
```

Object values are transformed to their string representations automatically with the **ToString** method when the filter criteria is set with the **SetFilterCriteria** method. Knowing this, we can shorten the previous statement as follows:

```
fc.AddSelectedValue(iGDropDownList1.Items[1]);
```

Pay attention to the fact that we are not using the **Text** property of the **iGDropDownListItem** object. It can be null in the general case, but null values are ignored while processing values from the **SelectedValues** list. The **iGDropDownListItem.ToString()** call guarantees that you get the actual text representation of the drop-down list item on the screen.

To filter a column with check box cells, use the **SelectedCheckStates** list of the corresponding **iGFilterCriteria** object. This list should contain **CheckState** values corresponding to the check states you want to filter by. Implying that the **System.Windows.Forms** namespace is already imported, we can write the following code snippet to filter rows only with checked check box cells:

```
iGFilterCriteria fc = new iGFilterCriteria();
fc.SelectedCheckStates.Add(CheckState.Checked);
iGAutoFilterManager1.ColAutoFilter("chk").SetFilterCriteria(fc);
iGAutoFilterManager1.ReapplyFilter();
```

Note that when you create a new **iGFilterCriteria** object to pass it to the **iGColAutoFilter**.**SetFilterCriteria** method, its **SelectAllItems** property is set to False by default. The value of this property reflects the state of the special "(Select All)" check box at the top of item list in the filter box. Setting the **SelectAllItems** property to True by default would mean that all items are selected and there is no sense to add any selected value for filtering. As a result, you would need to set this property to False every time while constructing a new filter criteria. The default value of False allows you to avoid doing this setting and makes your code cleaner a little bit.

29.6.3. Reading Filter Criteria

To retrieve the filter criteria for a particular column, use the **iGColAutoFilter**.**GetFilterCriteria** method like in the code below:

```
iGFilterCriteria fc =
iGAutoFilterManager1.ColAutoFilter(3).GetFilterCriteria();
```

The returned **iGFilterCriteria** object is the representation of the currently set filter criteria for the column. You can analyze the values of its main properties used to define filter criteria, such as

SelectedValues and **CustomConditions**, to know what values or custom conditions were set by the users.

Pay attention to the **SelectAllItems** property in the returned **iGFilterCriteria** object. It contains a **Boolean** value indicating whether all items are selected, which reflects the state of the special "(Select All)" check box at the top of item list in the filter box. If you read the current filter criteria with the **GetFilterCriteria** method and the **SelectAllItems** property equals True, the **SelectedValues** and **SelectedCheckStates** lists in the returned **iGFilterCriteria** object will be empty. There is no sense in filling these lists with all available items if they all are selected, especially if the value list contains thousands of values, and the **GetFilterCriteria** method simply leaves these lists empty for faster execution.

29.7. Filtering Tree Grids

Filtering tree grids differs from filtering traditional grids because rows have hierarchical relations. When a tree grid row matches the filter criteria, all its parent and child rows must also be displayed to maintain the tree's structural integrity, regardless of whether those related rows match the filter. AutoFilterManager allows you to filter tree grids according to this rule.

Let's consider a tree grid displaying some human vehicles and their main parts in hierarchy relations:

Row	Item	Level
1	Bicycle	0
2	Wheel	1
3	Tube	2
4	Rim	2
5	Spoke	3
6	Frame	1
7	Brake	1
8	Caliper	2
9	Rubber Pad	2
10	Brake Wire	2
11	Seat	1
12	Unicycle	0
13	Wheel	1
14	Tube	2
15	Rim	2
16	Spoke	3
17	Seat	1
18	Skateboard	0
19	Deck	1
20	Truck	1
21	Wheel	1

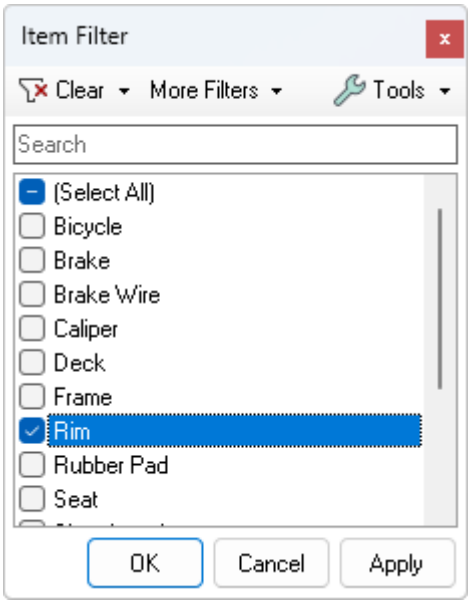
The data set for this grid is created with the following method:

```
private DataTable GetDataTable()
{
  var dt = new DataTable();

  dt.Columns.Add("Row", typeof(int));
  dt.Columns.Add("Item", typeof(string));
  dt.Columns.Add("Level", typeof(int));

  dt.Rows.Add(1, "Bicycle", 0);
  dt.Rows.Add(2, "Wheel", 1);
  dt.Rows.Add(3, "Tube", 2);
  dt.Rows.Add(4, "Rim", 2);
  dt.Rows.Add(5, "Spoke", 3);
  dt.Rows.Add(6, "Frame", 1);
  dt.Rows.Add(7, "Brake", 1);
  dt.Rows.Add(8, "Caliper", 2);
  dt.Rows.Add(9, "Rubber Pad", 2);
  dt.Rows.Add(10, "Brake Wire", 2);
  dt.Rows.Add(11, "Seat", 1);

  dt.Rows.Add(12, "Unicycle", 0);
  dt.Rows.Add(13, "Wheel", 1);
  dt.Rows.Add(14, "Tube", 2);
  dt.Rows.Add(15, "Rim", 2);
  dt.Rows.Add(16, "Spoke", 3);
  dt.Rows.Add(17, "Seat", 1);

  dt.Rows.Add(18, "Skateboard", 0);
  dt.Rows.Add(19, "Deck", 1);
  dt.Rows.Add(20, "Truck", 1);
  dt.Rows.Add(21, "Wheel", 1);

  return dt;
}
```

The returned **DataTable** is used in the form's **Load** event this way:

```
private void Form1_Load(object sender, EventArgs e)
{
  iGFillWithDataOptions opts = new iGFillWithDataOptions
  {
    RowLevelCol = "Level",
    AddRowLevelCol = true,
    AddTreeButtons = true
  };
  iGrid1.FillWithData(GetDataTable(), opts);

  iGrid1.TreeCol = iGrid1.Cols["Item"];
  iGrid1.TreeLines.Visible = true;

  iGrid1.Cols.AutoWidth();
}
```

Let's find all vehicles containing rims using the interactive filter box that drop downs when we click its button in the column header of the Item column:



The result is on the screenshot below:

Row	Item	Level
1	Bicycle	0
2	Wheel	1
4	Rim	2
5	Spoke	3
12	Unicycle	0
13	Wheel	1
15	Rim	2
16	Spoke	3

AutoFilterManager made visible not only the rows that match the filter (rows 4 and 15), but also their parent and child rows.

This tree grid filtering mode is enabled automatically if the **TreeCol** property of iGrid is not null. Otherwise the traditional grid filtering mode is used. In our sample, we would see only rows 4 and 15 after applying our filter if the **TreeCol** property was not set with the following statement in the code shown above:

```
iGrid1.TreeCol = iGrid1.Cols["Item"];
```

Pay attention to the fact that if the tree column is not set explicitly, iGrid automatically displays cell texts with level indents in the first visible column (see the Tree Grid Basics topic for more details). However, even if you see a tree structure in the first visible column, this does not mean AutoFilterManager will use the tree grid filtering mode described above. To enable it, you must set the first column as the tree column explicitly:

```
iGrid1.TreeCol = iGrid1.Cols[0];
```

29.8. AutoFilterManager and the Target Grid

When AutoFilterManager is attached to a grid, it affects some members of the target grid to provide the autofilter functionality. It is not advised to change these members when AutoFilterManager is attached to the grid as it may cause unpredictable behavior and prevent the add-on from working properly. These members are:

1. To display the filter button in a column, AutoFilterManager creates an object of an internal **iGDropDownFilterBox** class that represents a column's filter box and stores it in the **DropDownControl** property of the corresponding column header object (see the **iGColHdr** class).
2. To show/hide filtered rows, AutoFilterManager changes the **VisibleFiltered** property of the iGrid row object (**iGRow**).

AutoFilterManager attaches its event handlers to some iGrid's events listed below:

1. iGrid's **ColsAdded** event is used to create filter buttons in the attached grid when new columns are created in it.
2. The **AfterCommitEdit** event of iGrid is used to notify AutoFilterManager about the moment when it needs to reapply filter after a cell has been edited.
3. To draw the filter button, AutoFilterManager adds its own event handler to the **CustomDrawColHdrComboButton** event.
4. The filter button's tooltip with the effective filter expression is constructed by AutoFilterManager in the **RequestColHdrElemControlToolTipText** event of the target grid.

You may have your own event handlers for the **ColsAdded** and **AfterCommitEdit** events of iGrid, and they will coexist with AutoFilterManager's event handlers without any problems. As for the other two events related to iGrid column headers, don't redefine them (or provide your own actions only for columns excluded from filtering). Fortunately this will not happen in the vast majority of real-world grids because there is no need in redefining the behavior of drop-down buttons in column headers if they are already used for the autofilter functionality.

30. PRINTING WITH THE PRINTMANAGER ADD-ON

30.1. Introduction to PrintManager

PrintManager is an add-on component implementing the printing and print-preview functionality for iGrid.NET. This component is available for an extra fee for iGrid.NET users.

The component allows you to control the look of the grid on the paper. You can print iGrid.NET in a plain black-and-white or colored 3D mode, and adjust the parameters of the paper (print margins, the size of the paper and its orientation). You can also enhance the resulting document by adding headers and footers with calculated fields (such as the total number of pages and the current printing page).

The central component in the PrintManager add-on is **iGPrintManager**. Instances of this class are used in applications to provide printing and print-preview functionality for iGrid.

The **iGPrintManager** component uses the standard .NET Framework **PrintDocument** class to implement its printing and print-preview functionality. You can get a reference to this object with the **Document** property of **iGPrintManager**. This object can be used to create your own print preview and page setup dialogs, set up the document parameters in code, etc.

30.2. PrintManager Basics

30.2.1. First Steps with PrintManager

A typical usage scenario for the PrintManager add-on is as follows:

1. Create an instance of the **iGPrintManager** component.
2. Customize its properties to adjust the look of iGrid on paper.
3. Specify the grid control to print or print-preview with the **Grid** property of **iGPrintManager**.
4. Call the **Print** or **PrintPreview** method of **iGPrintManager** the print or print-preview the target grid.

You can create an instance of the **iGPrintManager** component and adjust its properties (Steps 1-3) at design time interactively in the Windows Forms Designer or solely from code.

In the case of Windows Forms Designer first you create and instance of the **iGPrintManager** component by dragging it from the Visual Studio Toolbox and dropping it onto your form, then select it and change its properties in the Properties pane. The Windows Forms Designer generates the initialization code automatically, and all you have to do to print or print-preview iGrid is to call the corresponding method of **iGPrintManager** in your code (Step 4) — for example:

```
private void buttonPrintGrid_Click(object sender, System.EventArgs e)
{
  iGPrintManager1.PrintPreview();
}
```

If you want to work with PrintManager solely from code, first you must add a reference to the PrintManager library to your project (in the case of the Windows Forms Designer such a reference is added automatically when you drop the **iGPrintManager** onto your form). You can a reference to the PrintManager library in the Solution Explorer: right-click the References item, chose the Add Reference... context menu item, select the PrintManager library implemented in TenTec.Windows.iGridLib.Printing.vX.Y.dll in the opened Add Reference dialog, and click the OK button. Now you can print the grid using a code snippet like the following one (Steps 1-4 above):

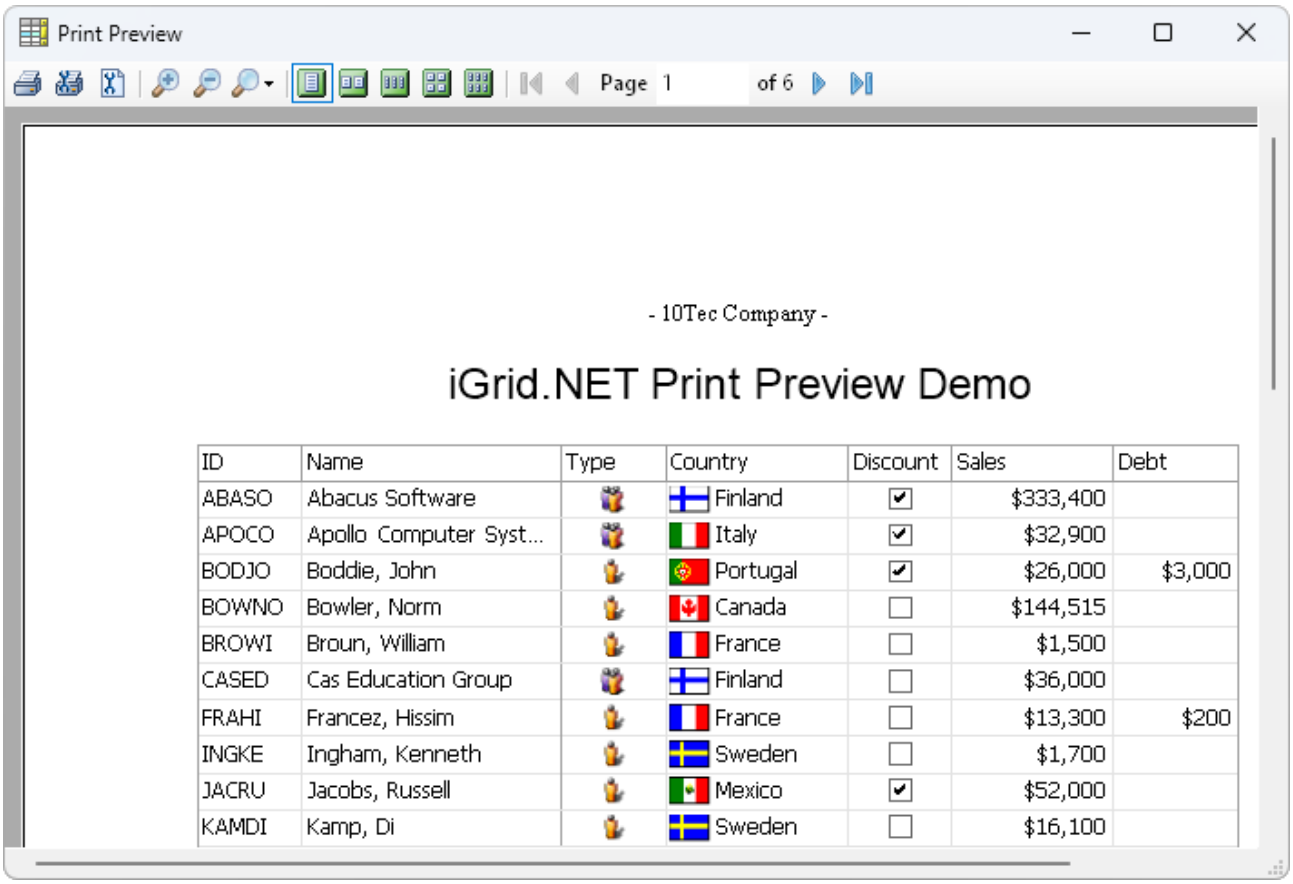
```
iGPrintManager pm = new iGPrintManager();
pm.PrintingStyle = iGPrintingStyle.ScreenView;
pm.Grid = iGrid1;
pm.Print();
```

The **Print** and **PrintPreview** methods of **iGPrintManager** have overloaded versions allowing you to specify the grid to print or print-preview as its argument. Using them, you can make the above code snippet a little bit shorter or print different iGrids with the same **iGPrintManager** component like this:

```
iGPrintManager pm = new iGPrintManager();
pm.PrintingStyle = iGPrintingStyle.ScreenView;
pm.Print(iGrid1);

// Print another iGrid with the same PrintManager:
pm.Print(iGrid2);
```

30.2.2. PrintManager Dialog Boxes

The **iGPrintManager** component implements several methods, the call of which leads to the display of the corresponding dialogs. The most significant of them is the **PrintPreview** method used to display the Print Preview dialog that demonstrates how the printed grid will look on paper. It is a rather standard dialog in which the user can print the document, adjust its layout properties and printer properties using the corresponding system dialogs, change the scale level and move between of the pages of the document:



The user can use the scroll bars around the document area to scroll the contents of the document. The document can also be scrolled with the mouse wheel. If the user rotates the mouse wheel without pressing any modifier keys, the document is scrolled in the vertical direction. If the user

holds down the SHIFT key while rotating the mouse wheel, the document is scrolled in the horizontal direction.

The mouse wheel has one more function in the Print Preview dialog. If the user holds down the CTRL key while rotating the mouse wheel, this action changes the scale level of the document in the dialog.

Two other methods of the **iGPrintManager** component that display dialogs are **PrintDialog** and **PageSetup**. They display the corresponding system dialogs in which the user can select the printer to print the grid and adjust main page parameters like paper size, orientation and margins.

30.3. Common Printing Tasks

30.3.1. Setting Page and Printer Parameters

When we print from a Windows Forms application using the standard approach, we use an object of the **PrintDocument** class. To adjust the page settings, we set the corresponding properties of its **DefaultPageSettings** object property.

In PrintManager you can retrieve this **PrintDocument** object from the **Document** property and change its setting the same way. For example, you need to print iGrid on paper in landscape and with the left/right page margins of 1 inch and top/bottom page margins of 0.5 inch. Taking into account the fact that in .NET printer margins are measured in hundredths of an inch, we could write the following code to get the job done:

```
var dps = iGPrintManager1.Document.DefaultPageSettings;
dps.Margins = new System.Drawing.Printing.Margins(100, 100, 50, 50);
dps.Landscape = true;
```

You can also control other printer settings through the **PrinterSettings** property of the **Document** property. For example, to print only the first two pages, use a code snippet like this:

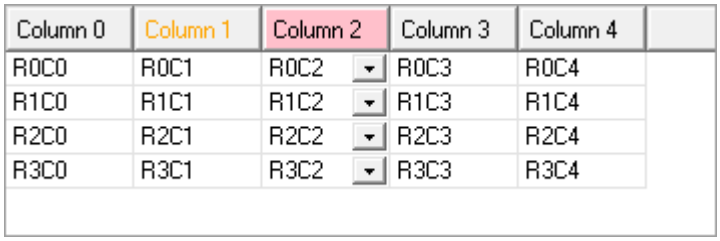
```
var ps = iGPrintManager1.Document.PrinterSettings;
ps.PrintRange = System.Drawing.Printing.PrintRange.SomePages;
ps.FromPage = 1;
ps.ToPage = 2;
ps.MinimumPage = 1;
ps.MaximumPage = 2;
```

30.3.2. Adjusting Look of Printed Grid

If we want to print a grid on the paper, the output can differ a lot from what we see on the screen. In one case we may want to print the grid "as is", keeping its color formatting and all elements like cell combo buttons on the paper. In another case we may want to save the ink or toner in our printer and print only the essential elements of the grid using minimal color formatting. The PrintManager component allows you to adjust the look of the grid on the paper to suit your needs.

The easiest way to adjust the look of the grid on the paper is to use the **PrintingStyle** property of PrintManager. This property accepts one of the 3 values from the **iGPrintingStyle** enumeration — **PlainView**, **ScreenView** or **Custom**.

We will demonstrate how these printing styles work while printing the following sample grid:



We specifically chose the classic 3D style for the grid because it contains quite a few colors and elements that can be turned on/off when printing:

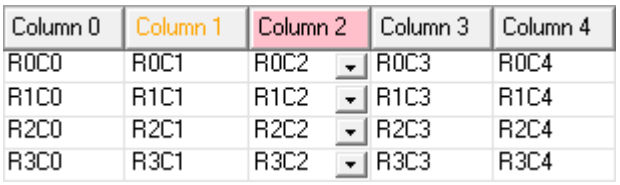
```
iGrid1.RenderStyle = iGRenderStyle.Classic;
```

The 2 colored column headers above will also be used to demonstrate how Plain View and Screen View work:

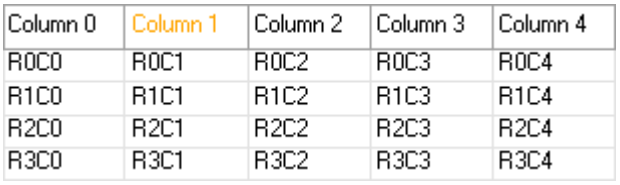
```
iGrid1.Header.Cells[0, 1].ForeColor = Color.Orange;
iGrid1.Header.Cells[0, 2].BackColor = Color.Pink;
```

Plain View and Screen View

If you want your grid to be printed as closer as possible to its screen representation keeping all color effects and elements, set the **PrintingStyle** property to **ScreenView**. This setting will produce the following output on the paper for our sample grid:



The **PlainView** setting will give the following output on the paper:



As you can see, PrintManager prints the grid as simple as possible without rich color effects. It achieves this goal by doing the following:

- All grid elements that may require a lot of ink/toner are replaced with their simplest equivalents on the paper. For example, 3D gray column headers are printed as non-filled rectangles with 1-pixel dark borders, OS-styled cell check boxes are drawn as non-filled squares with a simple tick image, etc.
- Only most significant parts of iGrid are printed. For example, cell combo or ellipsis button are not present on the paper.

If the plain view is used, column headers are 'simplified' for the paper if required depending on their on-screen style. This happens for the 3D style, but not for the flat or OS styles because they already have a plain flat look. The simplified column headers are represented with non-filled rectangles whose borders are made of the corresponding vertical and horizontal header grid lines defined in the **Header**.**HGridLinesStyle** and **Header**.**VGridLinesStyle** properties of iGrid. This allows you to adjust the border of plain column headers on the paper (color, width and dash style) with the corresponding properties for the screen elements. The same concerns row headers in the plain view — with the only difference that the corresponding grid line properties of the **iGrid**.**RowHeader** object are used.

The default value of the **PrintingStyle** property of PrintManager is **PlainView**. It generates the simplest look of the grid on the paper and saves the ink or toner of the printer.

Custom View and related properties

When you set the **PrintingStyle** property to **PlainView** or **ScreenView**, two other flag properties of PrintManager are changed respectively — **PrintingOptions** and **PrintCellControls**. They contain combinations of flags allowing you to adjust every elementary option related to grid look on the paper.

The **PrintingOptions** properties accepts flags from the **iGPrintingOptions** enumeration. These flags control the rendering of cells, column headers and row headers on the paper. They specify whether the corresponding elements will use their on-screen system style and will be filled with background color. For example, the **DrawSystemStyledHeader** flag specifies whether column headers will be drawn using the current system style like 3D, and the **FillColHdrBackground** flag specifies whether column headers on the paper will be filled with their effective background colors.

The **PrintCellControls** property accepts flags from the **iGPrintCellControls** enumerations. They specify which cell and column header elements, such as combo buttons and ellipsis buttons, will be printed. For example, the **ComboButton** flag specifies whether cell combo buttons will be printed.

When **PrintingStyle** is set to **PlainView**, **PrintingOptions** and **PrintCellControls** are set to **iGPrintingOptions**.**None** and **iGPrintCellControls**.**None** respectively turning off rich color effects and printing of non-essential grid elements. If **PrintingStyle** is set to **ScreenView**, **PrintingOptions** and **PrintCellControls** are set to **iGPrintingOptions**.**All** and **iGPrintCellControls**.**All** respectively, enabling all color effects and printing of all grid elements on the paper.

If only some flags (but not all) are set in the **PrintingOptions** and/or **PrintCellControls** properties, the **PrintingStyle** property is automatically set to **Custom** indicating that iGrid will have a custom look on the paper.

Below is an example demonstrating how to print our sample grid using a custom printing style in which cell combo buttons are absent and column headers are filled with their background color but not use their system style (3D in our case) on the paper:

Column 0	Column 1	Column 2	Column 3	Column 4
R0C0	R0C1	R0C2	R0C3	R0C4
R1C0	R1C1	R1C2	R1C3	R1C4
R2C0	R2C1	R2C2	R2C3	R2C4
R3C0	R3C1	R3C2	R3C3	R3C4

The corresponding PrintManager settings are below:

```
iGPrintManager1.PrintingOptions = iGPrintingOptions.FillColHdrBackground;
iGPrintManager1.PrintCellControls = iGPrintCellControls.None;
```

This visual effect for the header is achieved by specifying only one of the two **iGPrintingOptions** flags related to rendering of column headers on the paper. We do not use the **iGPrintingOptions**.**DrawSystemStyledHeader** flags to render column headers using the plain flat look, but use the **iGPrintingOptions**.**FillColHdrBackground** flag to fill those flat column header rectangles with their background colors.

Advanced color settings for printing

PrintManager provides several special color properties you can use to change the look of your grid on the paper. First of all, these are **GridLineColor**, **TreeLineColor** and **GroupBoxColHdrBorderColor**. Their values are used when PrintManager draws the corresponding parts of iGrid. By default all these color properties are set to **Color**.**Empty**, and PrintManager renders the corresponding elements exactly as they appear on the screen. But when

you specify a particular color in one of these properties, the corresponding elements will be rendered using the specified color. Note that the **GridLineColor** property is one color property for all grid lines in iGrid: if specified, its value overrides the colors of grid lines for cells, column headers and row headers.

The on-screen border of iGrid is not printed, but PrintManager provides one more special color property that allows you to draw a surrounding border for iGrid on the paper. You can print a 1-pixel solid border of a particular color around iGrid with the help of the **OutlineColor** property of PrintManager if you assign the desired color to this property.

30.3.3. Configuring Grid Layout on the Paper

To force PrintManager to print all columns on one page, set the **FitColsOnPage** property to **ResizeCols** or **ScaleGrid**. The default value of the **FitColsOnPage** property is **None**, which means that if the page width is not enough to print all columns, the last columns will be printed on another page. The **PrintOrder** property is used to control where to place these columns — on the next printed page, or after all first columns are printed.

The **PrintGridHeader** property specifies the pages on which PrintManager repeats the grid header area containing column headers. This property accepts the values of the **iGPrintGridHeader** enumeration: **OnEveryPage**, **BeforeFirstRow**, and **Never**. The default value is **OnEveryPage**, which causes the header area to be printed on every page. **BeforeFirstRow** causes it to be printed only on the first page or pages containing the first visible grid row. **Never** disables printing of the header area.

The **PrintGroupBox** property of PrintManager is used to specify whether the group box is printed. Similar to the grid header area, the group box can be omitted, printed only on the first page, or printed on all pages of the document.

The Boolean **PrintRowHeader** property allows you to specify whether the row header area is printed. Note that the row header area is printed only before the first visible column and thus can be turned on or off in contrast to the 3 layout options available for the header area and the group box above it.

One more useful layout setting can be set with the help of the **ForceExpandGroups** property. Your users can collapse/expand group rows in the grid and leave them each in its own collapsed state. If you want to print the grid as if its all group rows were expanded regardless of their actual state, set the **ForceExpandGroups** property to True. This setting has the same effect for trees in iGrid because the same row hierarchy system is used for them.

30.3.4. Adding Printable Headers and Footers

A document printed by PrintManager has the following 4 standard parts:

- Document header. It is printed once before all other parts on the first page.
- Document footer. It is printed once after all other parts on the last page.
- Page header. It is printed as the top of every page.
- Page footer. It is printed as the bottom of every page.

Every part represents a rectangular area which stretches from the left to the right edge of the page, and the general term "band" is also used instead when we talk about these parts of the document.

In PrintManager, you access to the bands listed above using the **DocumentHeader**, **DocumentFooter**, **PageHeader**, **PageFooter** properties respectively. Each property returns an instance of the **iGPageBand** class used to represent them.

The **iGPageBand** class contains three sections: left, middle, and right. Each of these sections is represented with the **iGPageBandSection** class and contains text with its own font. So you can separately specify the text that will be aligned to the left, middle, and right sides of a band. Two special fields, %[PageNumber] (current page number) and %[PageCount] (document page count), can be used in the section texts. They are replaced with the effective values when printing.

Below is an example of how to specify a document title centered horizontally and print page counter at the right bottom corner of every page using custom fonts:

```
var headerMiddle = iGPrintManager1.DocumentHeader.MiddleSection;
headerMiddle.Font = new Font("Arial", 18f);
headerMiddle.Text = "Annual Sales Report";

var footerRight = iGPrintManager1.PageFooter.RightSection;
footerRight.Font = new Font("Tahoma", 9.75f);
footerRight.Text = "Page %[PageNumber] Of %[PageCount]";
```

The **iGPageBand** class provides you with the **Indent** property to adjust the gaps between the band sides and page margins or other document elements. The indents are specified in the display coordinates (1 display pixel = 1/100 inch).

To add other custom graphics elements to the page bands (such as lines, logo, etc.), use the technique described in the Adding Custom Contents to Headers/Footers topic.

30.4. Advanced Features of PrintManager

30.4.1. Adjusting User Interface

PrintManager allows you to customize the window state of the print-preview window and the zoom factor selected in it by default. It can be done with the help of the **PrintPreviewSettings** compound property of the **iGPrintPreviewSettings** type. For example, the following settings can be used to open the print-preview window in the maximized state with the default zoom factor set to 200%:

```
iGPrintPreviewSettings pps = iGPrintManager1.PrintPreviewSettings;
pps.WindowState = FormWindowState.Maximized;
pps.Zoom = iGPrintPreviewZoom.Percent200;
```

All texts displayed by PrintManager (tooltips for controls of the print-preview window, warnings, etc.) can be changed/localized using the **UIStrings** object property.

30.4.2. Adding Custom Contents to Headers/Footers

Custom drawing in headers and footers

PrintManager allows you to use custom drawing to print your own headers and footers for the whole document or each page. You can draw your own contents in these bands in addition to the standard drawing or instead of it. These abilities allow you to implement such things as adding a logo to every page, using a custom background (say, a gradient fill) for page bands, drawing additional graphics elements like lines, etc.

Custom drawing is performed by handling the following 4 events: **CustomDrawDocumentHeader**, **CustomDrawDocumentFooter**, **CustomDrawPageHeader**, **CustomDrawPageFooter**. All these events share the same signature (see the **iGCustomDrawPageBandEventHandler** delegate) and expose identical properties in their event arguments:

- read-only **Graphics** — the **Graphics** object to draw on;
- read-only **Bounds** — the **Rectangle** structure to draw in;
- read-only **PageNumber** — the number of the current page;
- read-only **PageCount** — the total number of pages in the document;
- read-write Boolean **DoDefault** — a flag to tell PrintManager whether to draw the default contents.

The last **DoDefault** parameter is passed by reference and allows you to prohibit the drawing of the default contents if required. This property is set to True by default, and the default contents are drawn after your custom contents.

The size and location of each band is determined based on the values of such properties of PrintManager as **DocumentHeader**, **DocumentFooter**, **PageHeader** and **PageFooter**. You may need to change the bounds of the required band if you wish to draw your own contents in addition to the specified text (for instance, you may print the current page number using the **PageFooter**.**LeftSection**.**Text** property), or you may need to specify your own band placement if you do not use any text and wish to draw it from scratch. In this case, use the corresponding event from these 4 events designated for this purpose: **CustomDrawDocumentHeaderGetBounds**, **CustomDrawDocumentFooterGetBounds**, **CustomDrawPageHeaderGetBounds**, **CustomDrawPageFooterGetBounds**. They all have the same signature (see the **iGCustomDrawPageBandGetBoundsEventHandler** delegate) and these properties in the events' data:

- read-only **Graphics** — the **Graphics** object to draw on;
- **Bounds** — the **Rectangle** structure to draw in;
- read-only integer **PageNumber** — the number of the current page;
- read-only integer **PageCount** — the total number of pages in the document;

The **Bounds** parameter is passed by reference and is used to specify the required placement of the band. By default it is populated using the corresponding parameters of the printed page, but you can increase the band's height or even shift the left edge in these events. Every *GetBounds event is raised before the corresponding drawing event (for instance, **CustomDrawPageHeaderGetBounds** is raised before **CustomDrawPageHeader**), and thus the result bounds you could modify are passed to the drawing event.

Pay attention to the fact that the page numeration (the **PageNumber** argument) starts from 0.

An example of page header modification

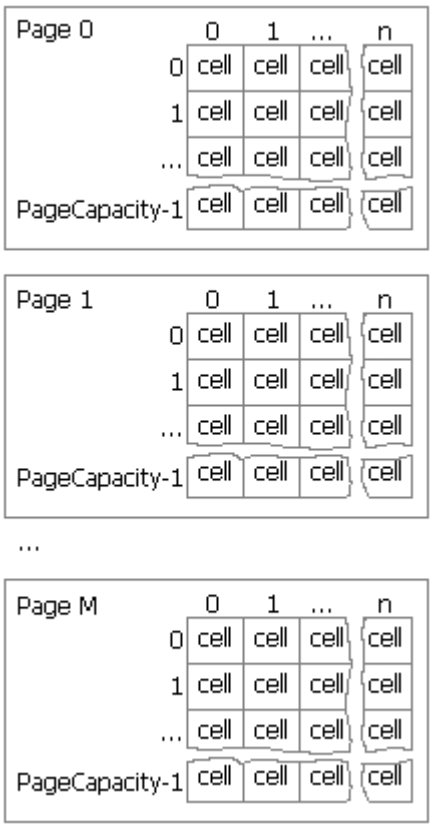
Let's assume you need to place your own image logo in the page header on the first page on a cute gradient background instead of the standard page header contents. The height of the standard page header band isn't enough for your custom contents, and it should be increased by 50 units. This is easily done with the help of the **CustomDrawPageHeader** and **CustomDrawPageHeaderGetBounds** events of PrintManager:

```
private void iGPrintManager1_CustomDrawPageHeaderGetBounds(
  object sender, iGCustomDrawPageBandGetBoundsEventArgs e)
{
  if (e.PageNumber == 0)
    e.Bounds.Height += 50;
}

private void iGPrintManager1_CustomDrawPageHeader(
  object sender, iGCustomDrawPageBandEventArgs e)
{
  if (e.PageNumber == 0)
  {
    e.DoDefault = false;

    using (LinearGradientBrush myGradientBrush = new LinearGradientBrush(
      e.Bounds, Color.AliceBlue, Color.AntiqueWhite, 45, false))
    {
      e.Graphics.FillRectangle(myGradientBrush, e.Bounds);
    }

    e.Graphics.DrawIconUnstretched(this.Icon, e.Bounds);
  }
}
```

31. OPTIMIZATION TIPS

31.1. Allocating Memory for iGrid Cells

iGrid uses its own memory pages to store cells. In simplified terms, you can think of each page as a two-dimensional array with a fixed number of rows and columns. The number of rows in a page is called "page capacity". Each row of the page corresponds to one row in iGrid. The number of columns is equal to the number of columns in iGrid plus 1. The extra column is used to store cells from the row text column, which is always present in iGrid:



When adding new rows, iGrid checks whether the existing pages can accommodate the existing and new rows. If not, iGrid creates new pages that will be sufficient to accommodate the entire set of rows. This approach allows us to improve the speed of adding new rows to the grid because iGrid does not reallocate existing cells and allocate memory by blocks if required only for new rows.

Note that the number of rows in the allocated pages is usually greater than the actual number of rows in iGrid. Unused rows in the pages will be used in the future when new rows are added, which also reduces the number of memory allocation operations and further speeds up iGrid.

When rows are deleted, iGrid does not delete the corresponding rows in the pages, but only marks them as deleted. These rows will be reused in the future to store data when new rows are added. This approach also helps reduce the number of memory allocation operations and speed up iGrid.

The default page capacity is 100. In other words, every page will hold 100 rows. This value can be changed with the **PageCapacity** property of iGrid. If you are working with huge grids containing thousands of rows and frequently adding/deleting rows, you can increase the standard page capacity to minimize the number of memory allocation operations and maximize the performance of your code. The optimal value depends on the grid usage scenario and, in complex scenarios, can be determined through experimentation. In the simplest case, when you know in advance the number of rows that iGrid will contain and this value will not change, it makes sense to set the **PageCapacity** property equal to this number of rows before adding them.

You can change the page capacity value before any pages have been created. If you try to do this after at least one page has already been created, iGrid will inform you of this by throwing an exception with a corresponding message. You can delete all pages from memory and enable page capacity changes after deleting all rows using the following two members of the **iGrid**.**Rows** collection: call its **Clear** method or set its **Count** property to 0.

Finally, here is one important recommendation regarding the order of actions when creating columns and rows in iGrid. Whenever possible, always create all columns before adding rows. If you create columns after creating rows, iGrid will have to recreate the pages with cells according to the new set of columns. In addition to allocating memory for each page, this will also involve copying data from old pages to new ones, which is a very resource-intensive process. Deleting columns after creating rows can also take quite a long time for the same reason.

31.2. Suppressing Screen Updates for Performance

When you change something in iGrid that requires an update on the screen (change a cell value or column width, sort or group data, etc.), by default, iGrid redraws its contents on the screen after each such operation. As mentioned in introductory topics, using the **BeginUpdate** and **EndUpdate** methods of iGrid allows you to disable unnecessary redrawing after each operation and thus significantly speeds up code execution. As a rule, code that performs batch updates is wrapped in calls to the **BeginUpdate** and **EndUpdate** methods:

```
iGrid1.BeginUpdate();

// perform batch changes

iGrid1.EndUpdate();
```

When you call **BeginUpdate**, iGrid disables any redraws. They resume after calling the **EndUpdate** method. It is very important to call the **EndUpdate** method to resume redraws, and therefore real-world applications may use a try/finally construct to ensure that **EndUpdate** is executed:

```
iGrid1.BeginUpdate();

try
{
  // perform batch changes
}
finally
{
  iGrid1.EndUpdate();
}
```

BeginUpdate calls are cumulative, and screen updating in iGrid is only enabled after all **BeginUpdate** calls have been completed with their corresponding **EndUpdate** calls. Below is a code snippet explaining this concept:

```
iGrid1.BeginUpdate();

iGrid1.CellValues[0, 0] = 111; // no updates on the screen

iGrid1.BeginUpdate(); // nested BeginUpdate() call

iGrid1.CellValues[0, 0] = 222; // no updates on the screen again

iGrid1.EndUpdate(); // still no updates because
                    // the very first BeginUpdate() is in effect

iGrid1.EndUpdate(); // iGrid is updated only after this EndUpdate()
```

If updates in iGrid are disabled, it continues to perform all other operations as usual — their results are simply not displayed on the screen. It is worth noting that the position of the scroll boxes on the scroll bars may also change at the same time. If this effect is undesirable, it can be disabled using the **AdjustScrollBarValuesRedrawOff** property of iGrid.

31.3. Fast Way to Get and Set Cell Values

The traditional way to read or write a cell property is to use the corresponding property of the **iGCell** class. One of them, **iGCell.Value**, is used to read/write cell values. Below is an example of code based on this property:

```
for (int i = 0; i < iGrid1.Rows.Count; i++)
  iGrid1.Cells[i, 0].Value = i;
```

iGrid provides you with an alternative way to read or write cell values — the **CellValues** property of iGrid. It is indexed the same way like the **Cells** array of iGrid. The previous code snippet can be rewritten with the **CellValues** property like this:

```
for (int i = 0; i < iGrid1.Rows.Count; i++)
  iGrid1.CellValues[i, 0] = i;
```

The main benefit of the **CellValues** property is that it works up to 2 times faster compared to the **iGCell.Value** property. The fact is that every access to a cell value using the **CellValues** property is done without creating an intermediate **iGCell** object returned by the **iGrid.Cells[·,·]** call. One more result of this approach is that memory is not filled with many useless **iGCell** objects, which led to less frequent garbage collection operations.

31.4. Retrieving Column and Row Keys

If you use string keys for iGrid columns and rows, you may need to retrieve these keys using numerical indexes of these objects. This often occurs in iGrid event handlers, for example:

```
private void fGridOutlook_CellDynamicContents(
  object sender, iGCellDynamicContentsEventArgs e)
{
  if (fGridOutlook.Cols[e.ColIndex].Key == "Flag")
  {
    // provide dynamic contents
  }
}
```

The example above demonstrates the straightforward way of doing this. In the expression **fGridOutlook.Cols[e.ColIndex].Key** we actually first retrieve the **iGCol** object representing the column with the **e.ColIndex** index using the traditional **iGrid.Cols[]** syntax, and then we read the value of the **Key** property of the returned **iGCol** object. The problem with this approach is that internally iGrid does not store **iGCol** objects for performance reasons and creates them dynamically every time when you access them using numeric indexes. If these operations are performed frequently, which is the case in events related to drawing and mouse processing, this can lead to more frequent garbage collection operations and overall performance degrade.

To avoid this problem, iGrid provides you with another way to obtain row and column keys without creating these objects. This can be done with the help of the **GetKey()** method implemented in the **iGrid**.**Cols** and **iGrid**.**Rows** collections. For example, the above event handler could be rewritten as follows using the **GetKey** method of the column collection:

```
private void fGridOutlook_CellDynamicContents(
  object sender, iGCellDynamicContentsEventArgs e)
{
  if (fGridOutlook.Cols.GetKey(e.ColIndex) == "Flag")
  {
    // provide dynamic contents
  }
}
```

31.5. Enumerating Collection Items in Reverse Order

All iGrid collections implement the **IEnumerable<T>** interface, so you can use the **Reverse()** LINQ method to enumerate items in reverse order as shown below:

```
foreach (iGRow row in iGrid1.SelectedRows.Reverse())
{
  // doing something with the row
}
```

However, the standard implementation of this **Reverse()** extension method first creates a copy of the whole collection and then iterates it in reverse order instead of simply enumerating the input collection in reverse order. It may result in creating an abnormally large number of objects in a short moment of time and increase the code execution time due to this unnecessary copy operation.

To solve this problem, iGrid provides the equivalent **ReverseOptimized()** method for all major collections that may contain a large number of objects. Among them are the following iGrid collections: **Cells**, **Cols**, **Rows**, **SelectedCells**, **SelectedRows**, and **MergedCells**. The iGrid implementation is free from the drawbacks described above and simply returns objects in reverse order ready for enumeration. According to tests performed, this optimized implementation can run 3 times faster than the standard one.

Use iGrid's **ReverseOptimized()** just like the standard LINQ **Reverse()** method, as shown below:

```
foreach (iGRow row in iGrid1.SelectedRows.ReverseOptimized())
{
  // doing something with the row
}
```

31.6. Batch Column/Row Operations

You can create new rows one-by-one and set the cell values in the created rows with the **Add** method of the **Rows** collection of iGrid, for example:

```
iGRow myRow = iGrid1.Rows.Add();
myRow.Cells[0].Value = "Text"; // value of cell in column 0
myRow.Cells[1].Value = 100;    // value of cell in column 1
myRow.Cells[2].Value = 200;    // value of cell in column 2
```

The iGrid control itself implements the **AddRow** method that can be used to add an empty row and optionally populate its cells with some values in one call. This method can help to make your code shorter and faster. The code sample above can be rewritten as follows using this method:

```
iGrid1.AddRow("Text", 100, 200);
```

If you need to create several rows in an empty iGrid, the best and fastest way to do this is to use the **Count** property of the **Rows** collection:

```
iGrid1.Rows.Count = 5;
```

If you are adding/inserting/deleting several consecutive rows, do so in a single statement using the Range-methods of the **Rows** collection (**AddRange**, **InsertRange**, **RemoveRange**). The following example shows how to insert 5 rows before the 10th row of the grid in one go:

```
iGrid1.Rows.InsertRange(9, 5);
```

The same optimization tips apply to columns. Use the **Count** property and the **AddRange**, **InsertRange**, **RemoveRange** methods of the **Cols** collection of iGrid to add, insert, and delete multiple consecutive columns. These members not only make your code more compact, but also allow you to avoid unnecessary memory allocation and reorganization operations described in the Allocating Memory for iGrid Cells topic.

32. ADVANCED TOPICS

32.1. Built-in Tooltips for Cells of All Kinds

If you have a grid cell with contents that aren't fully visible on the screen, iGrid automatically displays the built-in tooltip window with the full text of the cell when you pause the mouse pointer inside the cell for a short period of time. Note that the size of the cell may be enough to display the full text without clipping, but iGrid is enough smart and displays the tooltip window even if the text is covered by the edge of the grid or another element. If the cell contains other items, such as image, and they are partially hidden, the standard tooltip window with the cell text is also displayed in this case.

When iGrid decides whether to display its built-in tooltip window for a cell, it raises the **RequestCellToolTipText** event and passes the tooltip text it is going to display in its **Text** parameter. This parameter is passed by reference, and you can change it here. For instance, you can break a long cell text onto several lines.

This event is raised for all cells, even for those that do not have truncated parts. This allows you to use the built-in tooltips to display your custom extended information for cells. The following simple example demonstrates how to display the row and column indexes for every cell in iGrid:

```
void iGrid1_RequestCellToolTipText(
  object sender, iGRequestCellToolTipTextEventArgs e)
{
  e.Text = String.Format("Row: {0}; Column: {1}", e.RowIndex, e.ColIndex);
}
```

You can use this approach to retrieve additional information for your cells from a database, etc.

If the tooltip should not be displayed for a cell, iGrid passes null (Nothing in VB) in its **Text** parameter. You can also use this if you wish to suppress the tooltips for some or all of your cells. To do that, simply set **Text** to null or an empty string. The following code suppresses the displaying of the built-in tooltips for all cells in iGrid:

```
void iGrid1_RequestCellToolTipText(
  object sender, iGRequestCellToolTipTextEventArgs e)
{
  e.Text = String.Empty;
}
```

If you need to know whether the cell text or another part of the cell is clipped at the moment and thus really requires a tooltip, use the **IsCellPartClipped** method of the cell (the **iGCell** object). iGrid uses the same function to check whether the cell has clipped parts and requires the default tooltip.

The **RequestCellToolTipText** event also allows you to adjust the period of time the tooltip remains visible if the mouse pointer is stationary on iGrid. This is done with the help of the **AutoPopDelay** integer property of the event arguments object. This property works exactly like the **AutoPopDelay** property of the WinForms **ToolTip** component. It specifies the period of time in milliseconds, and its default value is 5000 (5 seconds). Below is an example how to set the cell tooltip display time to 10 seconds:

```
private void iGrid1_RequestCellToolTipText(
  object sender, iGRequestCellToolTipTextEventArgs e)
{
  e.AutoPopDelay = 10000;
}
```

You can also adjust column header tooltips and footer cell tooltips the same way. Use the **RequestColHdrToolTipText** and **RequestFooterCellToolTipText** events for that.

32.2. Size Box and Size Grip

A form can have a special square area in its bottom-right corner allowing the user to resize the form. This area is also known as "size box" or "resize corner". As a rule, this area displays a special triangle glyph called "size grip" indicating that the user can drag it to resize the form. Some standard WinForms controls like **StatusBar** show the size grip automatically if they occupy the bottom-right corner of the form. iGrid also automatically draws the size grip if its bottom-right corner is at the bottom-right corner of a resizable form and both scroll bars are visible:

The size box area with the size grip is displayed in the bottom-left corner of iGrid in right-to-left mode.

Pay attention to the fact that in the terms of iGrid "size box" is not the same as "size grip". The size box area always appears at the intersection of scroll bar continuations, but the size grip is drawn inside it only when the form can be resized.

Note also that the size box area is not visible if at least one of the scroll bars is hidden. If you want to provide your users with the ability to resize the form using iGrid's size grip, you need to make both scroll bars always visible. This can be done by setting the **Visibility** property of the scroll bars to **iGScrollBarVisibility**.**Always**.

The **IsPointOverSizeBox** method of iGrid can be used to determine whether a point belongs to the size box area. The **checkSizeGripVisible** parameter of this method allows you to determine whether the size grip is currently drawn.

You can redefine the drawing of the size box area and size grip by implementing the **DrawSizeBox** method of the **IiGControlPaint** interface in your custom control painter class.

32.3. Disabling Whole iGrid

Many if not all Windows Forms controls provide you with the Boolean **Enabled** property you can use to disable operations in a control. As a rule, a disabled control change its appearance to indicate that it is disabled — for example, it turns gray.

iGrid also implements the **Enabled** property. Its default value is True, but you can change it to False to disable any operations in iGrid. If iGrid is disabled, the contents of its normal cells, footer cells and column headers becomes gray. This effect is applied only to the foreground contents, such as cell texts and images, but not cell backgrounds.

The color used to render texts in normal cells, footer cells and column headers in a disabled iGrid is stored and can be adjusted with the **ForeColorDisabled** property of iGrid. Its default value is **SystemColors**.**GrayText**, which creates the effect of grayed text in a disabled iGrid with other default settings.

32.4. Keyboard Input Processing

When iGrid is in browse mode, you can use the standard **KeyPress** and **KeyDown** events to process keyboard input. To prohibit the default action assigned to a specific key, handle the **KeyPress** or **KeyDown** event and set the **Handled** field of the event arguments to True.

The following example shows how to redefine the behavior of the TAB key in iGrid. By default, pressing TAB moves input focus to the next cell. When the last cell is reached, input focus remains in the grid. The following event handler changes this behavior and moves input focus to the next control on the form when the last cell in the grid is reached:

```
private void iGrid1_KeyDown(
  object sender, System.Windows.Forms.KeyEventArgs e)
{
  if (e.KeyCode == Keys.Tab)
  {
    // Save the current cell.
    int myOldCurCellRowIndex, myOldCurCellColIndex;
    if (iGrid1.CurCell != null)
    {
      myOldCurCellRowIndex = iGrid1.CurCell.RowIndex;
      myOldCurCellColIndex = iGrid1.CurCell.ColIndex;
    }
    else
    {
      myOldCurCellRowIndex = -1;
      myOldCurCellColIndex = -1;
    }

    // Try to move the current cell to the next/previous column.
    if (e.Shift)
      iGrid1.PerformAction(iGActions.GoPrevCol);
    else
      iGrid1.PerformAction(iGActions.GoNextCol);

    // Get the current cell row and column indexes.
    int myCurCellRowIndex, myCurCellColIndex;
    if (iGrid1.CurCell != null)
    {
      myCurCellRowIndex = iGrid1.CurCell.RowIndex;
      myCurCellColIndex = iGrid1.CurCell.ColIndex;
    }
    else
    {
      myCurCellRowIndex = -1;
      myCurCellColIndex = -1;
    }

    // If the current cell has not been changed,
    // move input focus to the next control.
    if (myCurCellRowIndex == myOldCurCellRowIndex &&
        myCurCellColIndex == myOldCurCellColIndex)
    {
      ProcessTabKey(!e.Shift);
    }

    e.Handled = true;
  }
}
```

When the user is editing a text cell, iGrid places a text box editor over the cell and all keyboard events are directed to it. As a result, the **KeyDown** and **KeyPress** events are not raised. In this

case you should use the special events like **TextBoxKeyDown** to process keyboard input. They are described in the Text Edit Events and the TextBox Property topic.

32.5. Common Settings for Search-as-Type Tools

iGrid provides you with the ability to use its search-as-type system in the following parts of its UI:

1. The search-as-type functionality for normal iGrid cells.
2. The search-as-type functionality for items of built-in drop-down lists.
3. The cell autocomplete feature that can be enabled for text editing.
4. The search-as-type functionality for the pick list in the AutoFilterManager filter box.
5. The search box above the pick list in the AutoFilterManager filter box.

The internal implementation of iGrid's search-as-type functionality uses the **StartsWith** and **IndexOf** methods of the .NET **String** class to perform search. The overloaded versions of these methods accepting the string comparison rule (a member from the **StringComparison** enumeration in .NET) are used at that. By default iGrid passes the **CurrentCultureIgnoreCase** option to the corresponding calls of these methods to use the string comparison rules specific for the current culture of the user, which is the expected behavior in the vast majority of cases. However, sometimes the user may need search that does not take into account the string comparison rules of their language. This can be the case for some European languages in which two symbols can represent one characters (so called digraphs). For example, the "sz" combination in the Hungarian language actually represents one character, but the user may want to process the characters "s" and "z" written one-by-one independently like in English. The **SearchAsTypeStringComparison** property of iGrid allows you to implement this behavior that is not culture-specific.

The **SearchAsTypeStringComparison** property accepts one of the values from the standard **StringComparison** enumeration. The default value is **CurrentCultureIgnoreCase**, which corresponds to the culture-specific search-as-type behavior described above. To use string comparison rules that do not specific for a particular culture, set this property to **InvariantCultureIgnoreCase**:

```
iGrid1.SearchAsTypeStringComparison =
  StringComparison.InvariantCultureIgnoreCase;
```

If we look at the example with the Hungarian language above, this setting will allow you to process the characters "s" and "z" independently for the Hungarian culture.

Note that the **StringComparison** enumeration contains such members as **CurrentCulture** and **InvariantCulture**. They can be used to provide case-sensitive search-as-type functionality in iGrid.

Pay also attention to the fact that the **SearchAsTypeStringComparison** property of iGrid affects all 5 search-as-type tools listed in the beginning of this section, not just the search-as-type functionality for normal grid cells.

32.6. Performing Typical User Actions from Code

iGrid provides you with the **PerformAction** method that allows you to emulate standard user actions from code (such as selecting the next column, the next row, the first column, etc.). The **iGActions** enumeration contains items for all user actions you can pass to the **PerformAction** method:

ACTION	DESCRIPTION
CollapseAll	Collapse all the groups.

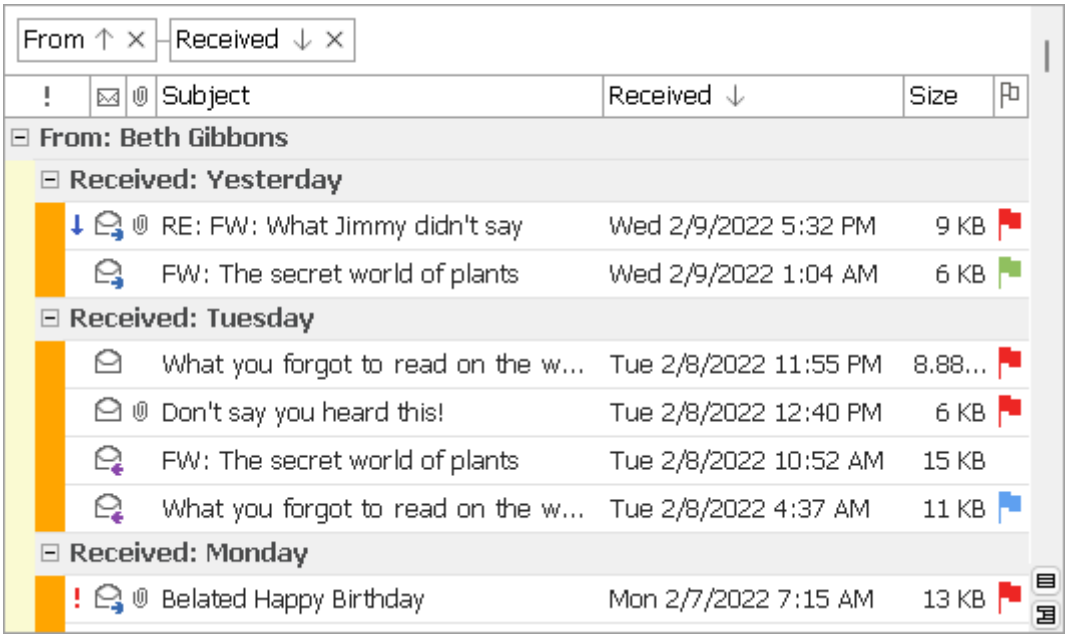
ACTION	DESCRIPTION
ExpandAll	Expand all the groups.
GoFirstCol	Move the current cell to the first column.
GoFirstRow	Move the current cell to the first row.
GoLastCol	Move the current cell to the last column.
GoLastRow	Move the current cell to the last row.
GoNextCol	Move the current cell to the next column.
GoNextPage	Move the current cell down a large distance.
GoNextRow	Move the current cell to the next row.
GoPrevCol	Move the current cell to the previous column.
GoPrevPage	Move the current cell up a large distance.
GoPrevRow	Move the current cell to the previous row.
SelectAllCells	Select all the cells.
DeselectAllCells	Deselect all the cells.
SelectAllRows	Select all the rows.
DeselectAllRows	Deselect all the rows.

The **DrawActionGlyph** method allows you to draw the predefined glyphs for the **CollapseAll**, **ExpandAll**, **GoFirstCol**, **GoLastCol**, **GoFirstRow**, **GoLastRow**, **GoPrevPage**, **GoNextPage**, **SelectAllCells**, **SelectAllRows**, **DeselectAllCells** and **DeselectAllRows** actions.

The **Action** property of the scroll bar custom button allows you to assign an action to it. If an action is assigned to a button, it will be performed when the button is clicked. The action's icon will also be automatically drawn on the custom button if the **DrawActionGlyph** method supports the action.

32.7. Custom-Drawn Level Indents

Drawing of grids with group rows and tree grids is based on the concepts of row levels and level indents inside rows (see the Row Levels and Level Indents topic). iGrid draws its standard level indents depending on the type of hierarchy (group rows or tree grids), but you can replace them with custom drawing if required. For example, you can draw a yellowish level indent for the first row hierarchy and an orange one for the second row hierarchy:



This is done with the help of the **CustomDrawLevelIndentPart** event.It is raised sequentially for the level indents in every row from left to right (right to left in right-to-left mode), enumerating these parts as 0, 1, 2, etc. The index of the currently drawn part is passed in the **PartIndex** property of the event arguments.

For example, the level indent area of the 5th row on the picture above consists of two parts. The first one (index 0) is filled with a sandy color, the second part (index 1) is filled with a dark orange color. The source code used to achieve this color effect is placed below:

```
void iGrid1_CustomDrawLevelIndentPart(
  object sender, iGCustomDrawLevelIndentPartEventArgs e)
{
  switch (e.PartIndex)
  {
    case 0:
      e.Graphics.FillRectangle(Brushes.LightGoldenrodYellow, e.Bounds);
      e.DoDefault = false;
      break;
    case 1:
      e.Graphics.FillRectangle(Brushes.Orange, e.Bounds);
      e.DoDefault = false;
      break;
    default:
      // Use the default drawing
      break;
  }
}
```

Note that if you redefine the drawing of a level indent, you should tell iGrid about that by setting the **DoDefault** parameter of the event arguments to False.

32.8. Objects as Drop-Down List Item Values

When basic data types such as strings or numbers are used as values for drop-down list items, internally iGrid uses a fast binary search algorithm to find the drop-down list item that corresponds to the current cell value. This is possible because all basic data types implement the standard .NET **IComparable** interface and its only **CompareTo** method. This feature allows iGrid to find the corresponding values in the list very fast even if you work with long lists of thousands of items.

However, if you use values of your own data type or one of the .NET data types that does not support **IComparable** (for instance, **IntPtr**), a traditional item-by-item linear search is used. To speed up the process for very long lists we recommend that you implement the **IComparable** interface in your objects if it is possible.

For instance, if you store instances of the following class in a drop-down list

```
private class CMyObject
{
  private string m_sText = string.Empty;

  public CMyObject(string sText)
  {
    m_sText = sText;
  }

  public override string ToString()
  {
    return m_sText;
  }
}
```

, it will work faster with the following addition that implements the **IComparable** interface:

```
private class CMyObject: IComparable
{
  private string m_sText = string.Empty;

  public CMyObject(string sText)
  {
    m_sText = sText;
  }

  public override string ToString()
  {
    return m_sText;
  }

  public int CompareTo(object obj)
  {
    CMyObject objTo = (CMyObject)obj;
    return String.Compare(this.m_sText, objTo.m_sText);
  }
}
```

32.9. Saving and Restoring Grid Layout

iGrid provides you with the ability to save and restore the column, group and sort layouts. This allows you to save the column order, their widths and visibility, the current sort and group criteria and restore these settings later. As a rule, this is done when the user closes a form to save the user preferences and restore them when the form is opened the next time.

This task is implemented with the following properties of the **LayoutObject** object property of iGrid:

PROPERTY	DESCRIPTION
Flags	Gets or sets a value that indicates which of the layout data to save and restore.
Text	Gets or sets a string which represents the layout information in the xml format.

The easiest way to save and restore a layout is to obtain the layout string from the **Text** property, save it to a text stream, and then load it later:

```
private void Form1_Closed(object sender, System.EventArgs e)
{
  ...
  StreamWriter myWriter = new StreamWriter("iGrid1Layout.xml", false);
  myWriter.Write(iGrid1.LayoutObject.Text);
  myWriter.Close();
  ...
}
private void Form1_Load(object sender, System.EventArgs e)
{
  ...
  if (File.Exists("iGrid1Layout.xml"))
  {
    StreamReader myReader = new StreamReader("iGrid1Layout.xml");
    iGrid1.LayoutObject.Text = myReader.ReadToEnd();
    myReader.Close();
  }
  ...
}
```

The **Text** property allows you to get or set a string representation of layout. The string is in the XML format and can be easily used as the contents of an XML node inside an XML file — for example, as a part of your application's setting file. The following example shows how to save and restore layout as a part of an application's settings:

```
private void Form1_Load(object sender, System.EventArgs e)
{
  ...
  //Load the application setting
  XmlDocument myXmlDocument = new XmlDocument();
  myXmlDocument.Load("Test.xml");
  ...
  //Restore the layout
  RestoreLayout(iGrid1, myXmlDocument.DocumentElement);
  ...
}
private void Form1_Closed(object sender, System.EventArgs e)
{
  //Create the application settings document
  XmlDocument myXmlDocument = new XmlDocument();
  XmlNode myRootNode = myXmlDocument.CreateElement("AppSettings");
  myXmlDocument.AppendChild(myRootNode);
  ...
  //Save the layout
  AddLayoutNode(iGrid1, myRootNode);
  ...
  //Save the application settings
  myXmlDocument.Save("Test.xml");
}
private void AddLayoutNode(iGrid grid, XmlNode node)
{
  XmlNode myNode = node.OwnerDocument.CreateElement("GridLayout");
  myNode.InnerXml = grid.LayoutObject.Text;
  node.AppendChild(myNode);
}
private void RestoreLayout(iGrid grid, XmlNode node)
{
  XmlNode myNode = node.SelectSingleNode("GridLayout");
  grid.LayoutObject.Text = myNode.InnerXml;
}
```

You can specify which data to save and restore with the **Flags** property. Its value is a combination of the **iGLayoutFlags** flags. The following table lists the items of this enumeration:

FLAG	DESCRIPTION
ColOrder	Save and restore the order of the columns.
ColVisibility	Save and restore the visibility of the columns.
ColWidth	Save and restore the width of the columns.
Grouping	Save and restore the grouping.
Sorting	Save and restore the sorting.
UseColKeys	Indicates that iGrid uses column keys to identify the columns, otherwise column indexes are used.

The value returned by the **Text** property contains the <Col> nodes. Each of these nodes stores information about one column. To identify a column, iGrid uses numeric indexes or string keys — depending on whether the **UseColKeys** flag is specified.

If the **UseColKeys** flag is not specified, the columns will be identified by their numeric indexes. If the **UseColKeys** flag is specified, all the columns should have unique keys. In the other case the layout may not be restored correctly. If a column has no key, its layout will not be saved.

Using column keys to save and restore layouts results in more robust code, especially when the grid structure changes — such as adding, removing, or reordering columns — between application versions. In contrast, relying on column indexes can lead to incorrect identification. Note that not all grids support string-based column keys, and that's why the **UseColKeys** flag is not specified by default.

When restoring a layout, the column order index specified in the layout string must exactly match the number of columns in the grid. If the index exceeds or equals the column count, the layout may not restore correctly. This rule also applies to the sort and group indexes.

32.10. User Interface Strings and Localization

iGrid and its add-ons allow you to change the strings the user can see in the UI. As a rule, this feature is used during the localization process, but you can also use it to change the default interface strings in English even if your application is not localized.

The core grid control includes several customizable interface strings. For example, the message displayed in the group box when column headers are absent, as well as the hints shown in the search window during search-as-type operations, can be modified. The grid add-ons, AutoFilterManager and PrintManager, have more strings to change/localize. These are tooltips for toolbar buttons, names of custom filter conditions, etc. All these interface strings that can be changed are grouped in the **UIStrings** object property in every component. See the **iGrid**.**UIStrings**, **iGAutoFilterManager**.**UIStrings**, **iGPrintManager**.**UIStrings** properties for more information.

Note that the standard WinForms designer localization technique enabled with the **Localizable** property of the form does not work for iGrid. This limitation is related to the internal infrastructure of the .NET Framework that does not support complex objects like iGrid columns or **UIStrings** objects. Unfortunately, Microsoft has no plans to fix this problem. Thus, if you need to localize iGrid, you can do that only from code at run time.

This can even be a more preferable approach for iGrid's **UIStrings** objects because you can define one method in the application to localize all iGrid interface strings instead of duplicating these strings manually in every form with iGrid(s) in the WinForms designer.

As for column captions and their widths, the localization approach provided by the WinForms designer may be non-productive in real-world applications. The fact is that column captions can vary greatly from language to language, and it will be a tedious work to set optimal column widths for every language manually at design time. We recommend that you call the **AutoWidth** method of the iGrid column collection (**iGrid**.**Cols**) at run time to automatically fit column widths based on their captions, current grid font, display pixel density, and so on.