10Tec iGrid.NET 11.0
What's New in Control

Contents

v11.0.0 | 2023-May-12 (Release)

AutoFilterManager: Creating Filter Criteria from Code

New AutoFilterManager Types

This release of the AutoFilterManager add-on provides the ability to create and read filter criteria from code. The new **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** type is a new class from the **TenTec.Windows.iGridLib.Filtering** namespace. It was introduced to specify item list values to filter by. It is defined as follows:

```
public class iGFilterItem
{
    public object Value;
    public string Text;

    public iGFilterItem(object value, string text)
    {
        Value = value;
        Text = text;
    }
}
```

Pay attention to the fact that the value to filter by is specified with its raw value in the native format (the **Value** property) and the corresponding string representation on the screen (the **Text** property). In the general case, the value can differ from its string representation for the user (for example, if cell values are formatted using a format string), and we must specify both.

The **CheckState** type in the table above is the **System.Windows.Forms.CheckState** enumeration from .NET Framework. The **iGFilterConditionsCombineMode** type is a new enumeration from the **TenTec.Windows.iGridLib.Filtering** namespace; it contains 2 elements – **And** and **Or**.

iGFilterCondition is also a new class from the **TenTec.Windows.iGridLib.Filtering** namespace. It was introduced to specify custom filter conditions and is defined as follows:

```
public class iGFilterCondition
{
    public iGFilterConditionOperator Operator;
    public string Parameter;
}
```

, where **iGFilterConditionOperator** is a new enumeration containing all possible custom filter operators:

```
public enum iGFilterConditionOperator
{
    Equals,
    DoesNotEqual,
    GreaterThan,
    GreaterThanOrEqualTo,
    LessThan,
    LessThanOrEqualTo,
    Contains,
    DoesNotContain,
    StartsWith,
    DoesNotStartWith,
    EndsWith,
    DoesNotEndWith
}
```

Setting Filter Criteria

The existing **iGColAutoFilter** class representing the autofilter functionality for an iGrid column was supplemented with 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 with 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 this instance 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)**. 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. If so, an even better version of the code above could be the following:

```
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.

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 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. It accepts values from the System.Windows.Forms.CheckState enumeration. Add the check states you want to filter by to the **SelectedCheckStates** list. 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.

Reading Filter Criteria

To retrieve the filter criteria for a particular column, use the new **GetFilterCriteria()** method of the **iGColAutoFilter** class:

```
public iGFilterCriteria GetFilterCriteria();
```

A call example is 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 we considered above (**SelectedValues**, **CustomConditions**, etc.) 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.

AutoFilterManager: Other Improvements

1. [Enhancement][Change][Code-Upgrade] AutoFilterManager provides the ability to save and restore filter as string data using the Save/Restore commands from the Tools menu in the filter box. From code this can be done by calling the **SaveFilterToMemory()**, **RestoreFilterFromMemory()**, **SaveFilterToFile()**, **RestoreFilterFromFile()** methods of the **iGAutoFilterManager** class or by reading/writing the **FilterAsBase64String** property of the **iGColAutoFilter** class. As a part of this process, the previous versions of AutoFilterManager serialized filter criteria to Base64 strings using the .NET BinaryFormatter class. This caused two serious problems. First, filter criteria saved in one version of AutoFilterManager could not be read by a later version of AutoFilterManager after upgrade because **BinaryFormatter** always saves and checks the full version number of the strong-named library the serialized types are originated from. Second, the **BinaryFormatter** class was marked as obsolete in .NET Core and is no longer recommended for usage for security reasons.

   To avoid all these compatibility and security problems, the filter criteria serialization engine was rewritten from scratch in this release of AutoFilterManager to save and read filters as traditional XML. This resulted in the following important changes:

- Filter criteria stored in autofilter files and strings created from the previous versions of AutoFilterManager cannot be used with the current version. If AutoFilterManager encounters an attempt to restore grid filter from an autofilter file created in the previous format, it will inform about that with the corresponding message box and will reject file importing.
- The **iGColAutoFilter.FilterAsBase64String** property based on the .NET **BinaryFormatter** class in the previous versions was replaced with the new **iGColAutoFilter.FilterAsXmlString** property. As the property name implies, now column filter criteria is returned and accepted as an XML string.

2. [New][Change][Code-Upgrade] AutoFilterManager provides the new event **FilterBoxOpened** to inform you about the moment when a filter box showed on the screen The event arguments object of this this event has the type **iGFilterBoxVisibilityChangedEventArgs** introduced in this release. Its properties contain values indicating the column for which the filter box was opened (integer **ColIndex**) and the new visibility state (Boolean **Visible**):

```
public class iGFilterBoxVisibilityChangedEventArgs : EventArgs
{
  public readonly int ColIndex;
  public readonly bool Visible;
}
```

   The type of the event arguments object of the existing **FilterBoxClosed** event was changed from **System.EventArgs** to **iGFilterBoxVisibilityChangedEventArgs** to provide detailed information about what filter box was closed – exactly like the new **FilterBoxOpened** event. If you have event handlers for the **FilterBoxClosed** event, you should upgrade your code. Change the event type from **System.EventHandler** to **iGFilterBoxVisibilityChangedEventHandler** and the type of the event arguments object from **System.EventArgs** to **iGFilterBoxVisibilityChangedEventArgs** (both AutoFilterManager types are defined in the **TenTec.Windows.iGridLib.Filtering** namespace).

   Note that the **FilterBoxOpened** and **FilterBoxClosed** events have the same type. This allows you to use the same method to process both events. The **Visible** property of the event arguments object can be used in this case to execute the corresponding part of the method body depending on the new visibility state of the filter box.

3. [Optimization] Changing properties of the **UIStrings** object of AutoFilterManager may have taken much time, especially for grids with tens of columns. This is explained by the nature of the filter box that is automatically adjusted for any change in the UI (a menu item's text is localized, the font is changed, etc.). And the more columns you had, the bigger execution time was.

   This functionality was greatly optimized in this release of AutoFilterManager, and it is no longer depends on the number of columns in iGrid. For the sake of testing, we changed the properties of the **UIStrings** object of an AutoFilterManager component attached to an iGrid with 15 columns on a pc with an Intel Core i7 2.6GHz processor and 32Gb of RAM. The code was executed about 10 seconds in the previous version. Now this work is done in 0,25 seconds on the same pc.

   The **UIStrings** object also provides the new **BeginUpdate()**/**EndUpdate()** methods to disable update of the filter box UI after every change in its properties. If you wrap code that changes **UIStrings** object with these method calls, you will get even faster execution – up to 10 times faster. The same test code from the previous paragraph required just 0,027 seconds for execution after wrapping it with the **BeginUpdate()**/**EndUpdate()** calls.

Row Visibility Infrastructure Update

1. [New][Code-Upgrade] iGrid rows represented with instances of the **iGRow** class provide the new Boolean **VisibleFiltered** property. This property is related to the internal infrastructure and was introduced for the built-in search-as-type functionality and the external AutoFilterManager component

to make rows visible or hidden depending on the current filter criteria. In the previous versions of iGrid the **iGRow.Visible** property was used for this purpose, but this didn't allow the developer to implement interfaces in which AutoFilterManager or iGrid's search-as-type window could be used in grids with invisible rows. Now this is possible due to the new **iGRow.VisibleFiltered** property.

If you used the **iGRow.Visible** property in the previous versions of iGrid to know whether a row is visible after filtering with AutoFilterManager or the search-as-type window, now you should use the **iGRow.VisibleFiltered** property instead.

As mentioned above, now you can build a grid in which you as a developer hide rows by setting their **Visible** property to False and the user can filter remaining visible rows using AutoFilterManager or the search-as-type window. If you want to find rows visible on the screen after the user applied filter criteria to such a grid, you should analyze the values of two properties – **iGRow.Visible** and **iGRow.VisibleFiltered**. Only if both rows are True, the corresponding row is visible on the screen.

New Features of the Footer Section

1. [New] If you specified an aggregate function for a footer cell to calculate the corresponding kind of totals, you can retrieve the calculated total with the new **Total** property of the **iGFooterCell** class. This property has the **Decimal** data type and allows you to use the calculated value in mathematical operations or comparison operators 'as is'.

2. [New] This release introduces the new **FooterTotalsUpdated** event raised after the values of aggregate functions in footer cells have been updated. It can be used to know the recalculated total values and/or format them (for example, mark negative totals with red).

3. [New] The following 3 new events allowing you to adjust the contents of footer cells dynamically were implemented: **FooterCellDynamicContents**, **FooterCellDynamicFormatting**, and **FooterCellDynamicStringFormat**. They work like their counterparts for normal cells (**CellDynamicContents**, **CellDynamicFormatting**, and **CellDynamicStringFormat**).

   These footer events provide not only the row and column indices in their arguments, but also allow you to retrieve the calculated totals for cells they are raised for with the **Total** property of the event arguments object. This property, like the new **iGFooterCell.Total** property, returns the calculated total as a **Decimal** value that can be used in comparison operators 'as is'.

   Having all this, you can easily implement various formatting options for totals in footer cells. For example, the following event handler will display all negative totals in red:

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

   Another example. If you do not want to display zeros in total cells, now you can simply make these cells empty with the following event handler:

```
private void iGrid1_FooterCellDynamicContents(
  object sender, iGFooterCellDynamicContentsEventArgs e)
{
    if (e.Total == 0)
        e.Text = String.Empty;
}
```

4. [Enhancement][Change] The **RecalculateTotals** event is no longer raised if redrawing in iGrid is off. If you used this event to calculate custom totals in your code, unneeded recalculation when redrawing is off will not happen.

FillWithData Method Family Extension

1. [New] This release of iGrid introduces the new **iGFillWithDataOptions** class to provide an alternative way to specify arguments of the **FillWithData()** method. The definition of the class is below:

```
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  = false;
    public int    DataStartRow    = 0;
    public int    DataRowCount    = int.MaxValue;
    public bool   CloseDataReader = true;
    public bool   AppendRows      = false;
}
```

iGrid provides the corresponding new overloaded version of the **FillWithData()** method in which you can specify data uploading options with an object of the **iGFillWithDataOptions** type:

```
public void FillWithData(object dataSource, iGFillWithDataOptions options)
```

This approach can free you from specifying default values for unused parameters and make your code shorter and clearer. For example, if you wanted to show only the first 10 rows of a data table in iGrid, in the previous version of iGrid you had to use the following overloaded version of **FillWithData()**

```
public void FillWithData(
  object dataSource, bool useCurColSet,
  string rowKeyCol, bool addRowKeyCol,
  string rowLevelCol, bool addRowLevelCol, bool addTreeButtons,
  int dataStartRow, int dataRowCount)
```

and specify values for all parameters – though actually only **dataSource** and **dataRowCount** were significant:

```
iGrid1.FillWithData(dataTable, false, null, false, null, false, false, 0, 10);
```

Now this task can be implemented simpler with the following code:

```
var opts = new iGFillWithDataOptions() { DataRowCount = 10 };
iGrid1.FillWithData(dataTable, opts);
```

If you have several calls of the **FillWithData()** method that use the same arguments, the new **iGFillWithDataOptions** class allows you to encapsulate them in one object and use it in all similar calls to **FillWithData()**.

Most field names of the **iGFillWithDataOptions** class correspond to the parameters of existing overloaded versions of the **FillWithData()** method. The remaining fields provide new features introduced in this release of iGrid. These are:

- 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 specifies whether the rows from the specified data source are added to the existing grid rows (True) or instead of them (False). If **AppendRows** equals True, the value of the **UseCurColSet** field is ignored and it is considered that iGrid uses the existing column set.

2. [New] In the previous releases of iGrid you could specify the range of data rows to upload into iGrid with the **dataSartRow** and **dataRowCount** parameters of the **FillWithData()** method only in one overloaded version of this method:

```
public void FillWithData(
  object dataSource, bool useCurColSet,
  string rowKeyCol, bool addRowKeyCol,
  string rowLevelCol, bool addRowLevelCol, bool addTreeButtons,
  int dataStartRow, int dataRowCount)
```

In many real-world situations the row key column or tree parameters are not specified (the **rowKeyCol, addRowKeyCol, rowLevelCol**, **addRowLevelCol**, **addTreeButtons** arguments). To simplify coding in these situations, this release of iGrid provides two more overloaded versions of the **FillWithData()** method that can be used to upload data row ranges:

```
public void FillWithData(
  object dataSource, int dataStartRow, int dataRowCount)

public void FillWithData(
  object dataSource, bool useCurColSet, int dataStartRow, int dataRowCount)
```

They can be used to upload a subrange of rows into iGrid without specifying unneeded parameter values in the only overloaded version of **FillWithData()** in the previous version of iGrid that provides this feature.

3. [New] In the previous versions of iGrid you could call the overloaded version of the **FillWithData()** method that provides the **dataRowCount** parameter to populate iGrid only with the columns from the data source by specifying 0 as the value of **dataRowCount**, for example:

```
iGrid1.FillWithData(dataSource, false, null, false, null, false, false, 0, 0);
```

Now there is a more elegant and self-descriptive way to perform the same task with the new **FillWithDataCols()** method that accepts only one parameter – **dataSource**:

```
iGrid1.FillWithDataCols(object 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 **FillWithDataCols()** method, which allows you to continue working with the data reader without reopening it again.

4. [Change] The **FillWithData()** method no longer scrolls the grid to start if possible (if the column set remains the same and/or rows are added after existing rows). This helps to maintain the scrolling position in situations when new portions of data are added to existing ones in iGrid.

Enhancements in Text Editing Events

1. [New] The event argument classes providing data for cell text editing related events (the iGrid events whose names start with "TextBox") are inherited from one new class called **iGTextBoxEventArgs**:

```
public class iGTextBoxEventArgs : EventArgs
{
    public readonly int RowIndex;
    public readonly int ColIndex;
}
```

This means that now you can retrieve the row and column indices of the cell a particular event was raised for using the **RowIndex** and **ColIndex** properties of the event arguments object. This changes nothing for the keyboard related events (**TextBoxKeyDown**, **TextBoxKeyPress**, **TextBoxKeyUp**, **TextBoxFilterChar**, **TextBoxTextChanged**), but adds this ability to the mouse related events (**TextBoxMouseMove**, **TextBoxMouseDown**, **TextBoxMouseUp**, **TextBoxMouseClick**, **TextBoxMouseDoubleClick**, **TextBoxMouseEnter**, **TextBoxMouseLeave**, **TextBoxMouseHover**). If you do not need to use this new feature in **TextBoxMouse*** event handlers, your code does not require any changes. But if you want to retrieve **RowIndex** and/or **ColIndex**, you should change the type of the event arguments in existing event handlers from **EventArgs** to **iGTextBoxEventArgs**. For example, if you had an event handler like

```
private void iGrid1_TextBoxMouseHover(object sender, EventArgs e)
{
    System.Diagnostics.Debug.WriteLine("TextBoxMouseHover");
}
```

, you can change it to the following one to output the row and column index of the edited cell:

```
private void iGrid1_TextBoxMouseHover(object sender, iGTextBoxEventArgs e)
{
    System.Diagnostics.Debug.WriteLine(
      $"TextBoxMouseHover: {e.RowIndex}, {e.ColIndex}");
}
```

2. [New] This release of iGrid introduces the following new event arguments class for the **TextBoxMouseMove**, **TextBoxMouseDown**, **TextBoxMouseUp**, **TextBoxMouseClick**, **TextBoxMouseDoubleClick** events:

```
public class iGTextBoxMouseEventArgs : iGTextBoxEventArgs
{
    public readonly MouseEventArgs MouseData;
}
```

This means that now you can retrieve information about the mouse in event handlers of those 5 events. These data are provided by the **MouseData** property that stores an instance of the

System.Windows.Forms.MouseEventArgs class used traditionally in the corresponding mouse events of WinForms controls. Pay attention to the fact that **iGTextBoxMouseEventArgs** inherits **iGTextBoxEventArgs**, i.e. the **RowIndex** and **ColIndex** properties are also available. To use the new features in existing event handlers, replace the **EventArgs** type of the event arguments object with **iGTextBoxMouseEventArgs** like in the following event handler:

```
private void iGrid1_TextBoxMouseDown(object sender, iGTextBoxMouseEventArgs e)
{
    System.Diagnostics.Debug.WriteLine("TextBoxMouseDown: " +
      $"RowIndex = {e.RowIndex}, ColIndex = {e.ColIndex}, " +
      $"X = {e.MouseData.X}, Y = {e.MouseData.Y}, " +
      $"Button = {e.MouseData.Button}");
}
```

3. [New] The **TextBoxEditStarted** event was implemented. It is raised right after the text editing of a cell started. Its event arguments are represented with an instance of the **iGTextBoxEventArgs** class.

4. [New] The **iGTextBoxKeyDownEventArgs** class that provides data for the **TextBoxKeyDown** event of iGrid implements the new Boolean **SuppressKeyPress** property. The functionality of this property is similar to KeyEventArgs.SuppressKeyPress in WinForms: you can set **SuppressKeyPress** to True to suppress the subsequent **TextBoxKeyPress** event. Setting **SuppressKeyPress** to True also implies that iGrid will not perform the action corresponding to the pressed key in the **TextBoxKeyDown** event, i.e. as if the related **DoDefault** property was set to False (similarly to WinForms).

5. [New] The **iGTextBoxTextChangedEventArgs** class, which is the base class for the event arguments of the **TextBoxTextChanged** event, provides quick access to the current cell text with the new **Text** property. It can be used instead of **iGrid.TextBox.Text** in handlers of the **TextBoxTextChanged** event.

6. [Enhancement] The **TextBoxTextChanged** event fires at the start of editing only if the cell text changes (for example, if the pressed key replaces the cell text). In the previous releases it always fired at the start of editing.

Other Improvements

1. [New] Now you can adjust the period of time iGrid built-in tooltips remain visible if the pointer is stationary on iGrid. This is done with the help of the new integer property called **AutoPopDelay** added to the event arguments object of the following events:

   - **RequestCellToolTipText**
   - **RequestColHdrToolTipText**
   - **RequestFooterCellToolTipText**
   - **RequestCellElemControlToolTipText**
   - **RequestColHdrElemControlToolTipText**

   The **AutoPopDelay** 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;
}
```

   Pay attention to the fact that this setting is applied only to normal cells but not to column headers and footer cells. You should use the **RequestColHdrToolTipText** and **RequestFooterCellToolTipText** event respectively to control the display time for column headers and footer cells.

   The **AutoPopDelay** value specified in an **RequestCellToolTipText** event handler is passed to the subsequent **RequestCellElemControlToolTipText** event handler if the mouse pointer enters the area of an elementary control within the same cell (if any). This time can be adjusted for the cell control

if required. The same works for the pair of the **RequestColHdrToolTipText** and **RequestColHdrElemControlToolTipText** events.

2. [New] This release of iGrid introduces the concept of the current column that can simplify coding in many cases. The current column is simply the column containing the current cell, and iGrid provides several new members to work with it for parity with similar members related to rows.

   You can access the current column as an **iGCol** object using the new **CurCol** property. It works similar to the existing **iGrid.CurRow** property. For example, if iGrid does not have the current cell, **iGrid.CurCol** returns Null.

   The current column can be set using the new **SetCurCol()** method with the following two overloads:

```
public void SetCurCol(int colIndex);

public void SetCurCol(string colKey);
```

   They also work like the existing **SetCurRow()** method for rows. For example, if the current column is changed with the **SetCurCol()** method, actually the position of the current cell within the current row is changed, and the like.

   The two new events, **CurColChangeRequest** and **CurColChanged**, fire when the user is about to change the current column and after this has happened. Similar to the **CurRowChangeRequest** event, **CurColChangeRequest** can be used to know the index of the column the user is going to make current and prohibit this change on the fly if required.

3. [Enhancement][Code-Upgrade] The existing **CurCellChanged**, **CurRowChanged** events and the new **CurColChanged** event provide new event arguments allowing you to know the indices of the previous and the new cell/row/column. These fields are named **OldRowIndex**, **OldColIndex**, **NewRowIndex**, **NewColIndex** and are used in the corresponding event arguments only if applicable (for example, **CurColChanged** provides only **OldColIndex** and **NewColIndex**).

   All these fields have the **Nullable<Int32>** type to indicate the situation when the current cell/row/column was or became absent: if so, the corresponding parameter does not have a value (returns Null). Below is an example of how to track the situation if the column containing the current cell has been deleted and iGrid no longer has the current cell (and column):

```
private void iGrid1_CurColChanged(object sender, iGCurColChangedEventArgs e)
{
    if (!e.NewColIndex.HasValue)
        System.Diagnostics.Debug.WriteLine("No current column");
}
```

   The types of the existing **CurCellChanged** and **CurRowChanged** events were changed from **EventHandler** to **iGCurCellChangedEventHandler** and **iGCurRowChangedEventHandler** to provide the new fields in the event arguments objects, which now have the types **iGCurCellChangedEventArgs** and **iGCurRowChangedEventArgs** respectively. To upgrade existing code to work with this release of iGrid, do the corresponding changes.

   For example, if a **CurCellChanged** event handler was attached in C# code with a statement like

```
iGrid1.CurCellChanged += new System.EventHandler(iGrid1_CurCellChanged);
```

   , it should be transformed into this one:

```
iGrid1.CurCellChanged +=
  new iGCurCellChangedEventHandler(iGrid1_CurCellChanged);
```

   The existing event handler method

```
private void iGrid1_CurCellChanged(object sender, EventArgs e)
{
   …
}
```

should be changed to this:

```
private void iGrid1_CurRowChanged(object sender, iGCurCellChangedEventArgs e)
{
   …
}
```

The corresponding transformation is even simpler for VB code in which the Handles keyword is used to attach event handlers. For example, the following event handler definition

```
Private Sub iGrid1_CurCellChanged( _
  ByVal sender As Object, ByVal e As EventArgs) _
  Handles iGrid1.CurCellChanged
```

Will be transformed into this:

```
Private Sub iGrid1_CurCellChanged( _
  ByVal sender As Object, ByVal e As iGCurCellChangedEventArgs) _
  Handles iGrid1.CurCellChanged
```

4. [New] iGrid implements the following 4 new methods to select and deselect ranges of cells and rows:

```
void SelectCellRange(
  int startRowIndex, int startColIndex,
  int endRowIndex, int endColIndex);

void SelectRowRange(
  int startRowIndex, int endRowIndex);

void DeselectCellRange(
  int startRowIndex, int startColIndex,
  int endRowIndex, int endColIndex);

void DeselectRowRange(
  int startRowIndex, int endRowIndex);
```

The **SelectCellRange()** method is almost equivalent to the following code snippet:

```
iGrid1.BeginUpdate();
for (int i = startRowIndex; i <= endRowIndex; i++)
{
    for (int j = startColIndex; j <= endColIndex; j++)
    {
        iGrid1.Cells[i, j].Selected = true;
    }
}
iGrid1.EndUpdate();
```

The differences are the followings:

a) When you access a cell to change its selection status using the **iGrid.Cells[,]** call, an instance of the **iGCell** class is created for every accessed cell. As a result, a big number of **iGCell** objects marked for garbage collection can remain in memory after executing such a loop. This does not happen when calling the **SelectCellRange()** method.

b) Every change of the **iGCell.Selected** property leads to raising of the **SelectionChanged** event. In the case of **SelectCellRange()** this event is fired only once at the end of operation.

The method implementation is optimized for faster execution of the selection operation compared to the loop above. One of the optimizations is that the selection status of every processed cell is not checked to determine whether to raise the **SelectionChanged** event for it. Calling of the **SelectCellRange()** method instead of the loop not only makes your code shorter but faster too. The method is about 40% faster than the equivalent loop.

All what is described above for the **SelectCellRange()** method is also applicable to the **SelectRowRange()** method.

5. [New] If iGrid is grouped by a column with the sort type set to **iGSortType.BySelected**, the text of group rows created for selected and non-selected cells can be adjusted/localized using the 2 new string properties of the **iGrid.UIStrings** object property – **GroupRowTextSelected** and **GroupRowTextNotSelected**. Their default values are "Selected" and "Not Selected" respectively, which corresponds to the texts displayed by iGrid in the previous versions.
6. [Optimization] Some internal algorithms of iGrid were optimized to provide faster updates after changing data, such as row and column sets. As a result, in many bulk update scenarios you may see 40% performance improvements.

Bug Fixes

1. [Fixed] iGrid allows you to have merged cells and group rows in one grid. If some of these elements were made non-selectable, the previous version of iGrid may have hung after pressing the LEFT key or the SHIFT+TAB combination to move the input focus to the previous cell. This problem has been fixed in this release. The bug fix also addresses the **iGActions.GoPrevCol** interactive action you can perform with the **PerformAction()** method from code or custom scroll bar buttons with this action assigned to them.
2. [Fixed] iGrid may have hidden rows incorrectly after assigning a row pattern with the **VisibleParentExpanded** property set to False to the **iGRow.Pattern** property.
3. [Fixed] The **InsertRange()** method of the row collection (**iGrid.Rows**) added rows to the end instead of inserting before the specified row.
4. [Fixed] Group rows may have not been created if iGrid had invisible rows and non-default sort types were used for columns.
5. [Fixed] iGrid did not update its constituent parts correctly after enabling/disabling visual styles in the application or after changing the OS theme, including switching to contrast themes.

Addendum: Tags used to classify changes

[New]	-	Completely new feature.
[Change]	-	Change in a member functionality or interactive behavior.
[Fixed]	-	Fixed bug or solved problem.
[Removed]	-	Member was completely removed.
[Enhancement]	-	Some functionality was enhanced.
[Optimization]	-	Performance was improved.
[Renaming]	-	Member was renamed.
[Code-Upgrade]	-	Source code for previous versions may require changes.