DataGridView Control

The DataGridView control is the new grid control for Windows Froms 2.0. It replaces the DataGrid control with an easy to use and extremely customizable grid that supports many of the features that are needed for our customers.

About this document:
This material should not be considered a complete coverage of DataGridView but it tries to capture the high-level features with some depth in specific areas.

This document is divided into about 5 logical sections starting with a feature and structure overview, followed by an overview of built-in column and cell types. Next is an overview of working with data, followed by an overview of specific major features. Lastly, a "best practice" section concludes the main part of this document.

Most sections contain a "Q & A" section that answers common questions regarding the specific feature or area covered by the section. Note that some questions are duplicated in multiple sections based upon the section relevancy. The question and answers with code samples/snippets are contained in this document's only appendix to make for a "one-stop shop" of code.

Note that most of the context of this document comes from the DataGridView control documentation presented in MSDN.

Contents

1 What is the DataGridView

With the DataGridView control, you can display and edit tabular data from many different kinds of data sources.
The DataGridView control is highly configurable and extensible, and it provides many properties, methods, and events to customize its appearance and behavior. When you want your Windows Forms application to display tabular data, consider using the DataGridView control before others (for example, DataGrid). If you are displaying a small grid of read-only values, or if you are enabling a user to edit a table with millions of records, the DataGridView control will provide you with a readily programmable, memory-efficient solution.

The DataGridView control replaces and adds functionality to the DataGrid control; however, the DataGrid control is retained for both backward compatibility and future use, if you choose. See below for details on the differences between the DataGrid and DataGridView controls.

1.1 Differences between the DataGridView and DataGrid controls

The DataGridView control provides numerous basic and advanced features that are missing in the DataGrid control. Additionally, the architecture of the DataGridView control makes it much easier to extend and customize than the DataGrid control.

The following table describes a few of the primary features available in the DataGridView control that are missing from the DataGrid control.

DataGridView control feature	Description
Multiple column types	The DataGridView control provides more built-in column types than the DataGrid control. These column types meet the needs of most common scenarios, but are also easier to extend or replace than the column types in the DataGrid control.
Multiple ways to display data	The DataGrid control is limited to displaying data from an external data source. The DataGridView control, however, can display unbound data stored in the control, data from a bound data source, or bound and unbound data together. You can also implement virtual mode in the DataGridView control to provide custom data management.
Multiple ways to customize the display of data	The DataGridView control provides many properties and events that enable you to specify how data is formatted and displayed. For example, you can change the appearance of cells, rows, and columns depending on the data they contain, or you can replace data of one data type with equivalent data of another type.
Multiple options for changing cell, row, column, and header appearance and behavior	The DataGridView control enables you to work with individual grid components in numerous ways. For example, you can freeze rows and columns to prevent them from scrolling; hide rows, columns, and headers; change the way row, column, and header sizes are adjusted; change the way users make selections; and provide ToolTips and shortcut menus for individual cells, rows, and columns.

The only feature that is available in the DataGrid control that is not available in the DataGridView control is the hierarchical display of information from two related tables in a single control. You must use two DataGridView controls to display information from two tables that are in a master/detail relationship.

1.2 Highlight of features

The following table highlights the DataGridView's major features. Further details about a feature can be found later in this document

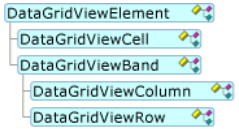
DataGridView control feature	Description
Multiple column types	The DataGridView control provides TextBox, CheckBox, Image, Button, ComboBox and Link columns with the corresponding cell types.
Multiple ways to display data	The DataGridView control can display unbound data stored in the control, data from a bound data source, or bound and unbound data together. You can also implement virtual mode in the DataGridView control to provide custom data management.
Multiple ways to customize the display and work with data	The DataGridView control provides many properties and events that enable you to specify how data is formatted and displayed. In addition, the DataGridView control provides multiple ways to work with your data. For example, you can: ▪ sort data with corresponding sort glyph ▪ enable selection modes by row, column or cell; multi-selection or single selection ▪ copy content to the clipboard in multiple formats including text, CSV (comma separated value) & HTML ▪ change the way users edit cell content
Multiple options for changing cell, row, column, and header appearance and behavior	The DataGridView control enables you to work with individual grid components in numerous ways. For example, you can: ▪ freeze rows and columns to prevent them from scrolling ▪ hide rows, columns, and headers ▪ change the way row, column, and header sizes are adjusted based upon size ▪ change the way users make selections ▪ provide ToolTips and shortcut menus for individual cells, rows, and columns ▪ customize the border styles of cell, rows and columns
Rich extensibility support	The DataGridView control provides the infrastructure to extend and customize the grid. For example, you can: ▪ handle custom painting events to provide a custom look and feel to the cells, columns and rows ▪ derive from one of the built-in cell types to provide additional behavior ▪ implement custom interfaces to create a brand new editing experience

2 Structure of DGV

The DataGridView control and its related classes are designed to be a flexible, extensible system for displaying and editing tabular data. These classes are all contained in the System.Windows.Forms namespace, and they are all named with the "DataGridView" prefix.

2.1 Architecture Elements

The primary DataGridView companion classes derive from DataGridViewElement.



The DataGridViewElement class provides a reference to the parent DataGridView control and has a State property, which holds a value that represents a combination of values from the DataGridViewElementStates enumeration.

2.2 Cells and Bands

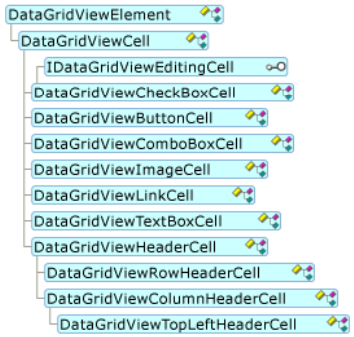
The DataGridView control comprises two fundamental kinds of objects: cells and bands. All cells derive from the DataGridViewCell base class. The two kinds of bands, DataGridViewColumn and DataGridViewRow, both derive from the DataGridViewBand base class.

The DataGridView control interoperates with several classes, but the most commonly encountered are DataGridViewCell, DataGridViewColumn, and DataGridViewRow.

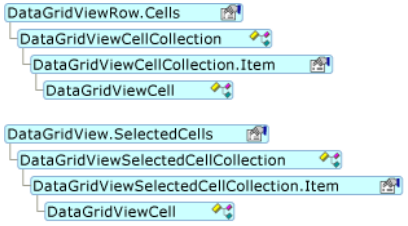
2.3 DataGridViewCell

The cell is the fundamental unit of interaction for the DataGridView. Display is centered on cells, and data entry is often performed through cells. You can access cells by using the Cells collection of the DataGridViewRow class, and you can access the selected cells by using the SelectedCells collection of the DataGridView control.

The DataGridViewCell class diagram



Cell Related Classes and Properties



The DataGridViewCell type is an abstract base class, from which all cell types derive. DataGridViewCell and its derived types are not Windoyws Forms controls, but some host Windows Forms controls. Any editing functionality supported by a cell is typically handled by a hosted control.

DataGridViewCell objects do not control their own appearance and painting features in the same way as Windows Forms controls. Instead, the DataGridView is responsible for the appearance of its DataGridViewCell objects. You can significantly affect the appearance and behavior of cells by interacting with the DataGridView control's properties and events. When you have special requirements for customizations that are beyond the capabilities of the DataGridView control, you can implement your own class that derives from DataGridViewCell or one of its child classes.

2.3.1 How a DataGridViewCell works

An important part of understanding the structure of the DataGridView is to understand how a DataGridViewCell works.

A Cell's Value

At the root of a cell is its value. For cells in a column that is not databound and the grid is not in virtual mode the cells actually store the value in the cell instance. For databound cells the cell doesn't "know" or keep the value is at all. Anytime the cell's value is needed the grid goes to the datasource and looks up the value for the column and row and returns that as the cell's value. In virtual mode this routine is very similar except the grid raises the CellValueNeeded event to get the cell's value. At the cell level, all of this is controlled via the DataGridViewCell::GetValue(...) method.

The data type for the cell's Value property by default is of type object. When a column becomes databound its ValueType property is set which causes each cell's ValueType to be updated. The value of the ValueType property is important for formatting.

Formatting for Display

Anytime the grid needs to know "how would this cell display" it needs to get its *FormattedValue*. This is a complex routine because formatting something on the screen usually needs to be converted to a string. For example, although you set a cell's value to the integer value of 155 when 155 needs to be displayed it has to become formatted for the display. The cells and column's FormattedValueType property determines the type that is used for display. Most columns use string, but the image and check box

cells\columns have different values. The DataGridViewImageCell and column use Image as the default FormattedValueType since its painting code knows how to display an image. A checkbox cell\column's FormattedValueType varies depending upon the value of ThreeState. At the cell level, all of this is controlled via the DataGridViewCell::GetFormattedValue(...) method.

By default, the DataGridView uses TypeConverters to convert a cell's value to its formatted value. Retrieving the proper TypeConverter is based upon the cell's ValueType and FormattedValueType properties.

For a cell, the FormattedValue is requested many times. Anytime the cell is painted or when a column needs to be autosized based upon the cell's content; the FormattedValue is even needed when determining if the mouse is over the cell content or not. Anytime the FormattedValue is required the DataGridView raises the CellFormatting event. This provides you with the opportunity to modify how the cell is formatted.

If a cell cannot retrieve its formatted value correctly it raises the DataError event.

Part of formatting a cell for display is understanding what the preferred size of the cell is. The preferred size is a combination of the cell's FormattedValue, any padding or additional display and the borders.

Painting the Display

After the FormattedValue is retrieved the cell's responsible for painting the cell's content. The cell determines the correct style to paint with (see the Styling section later in this document) and paints the cell. It is important to note that if a cell does not paint itself then nothing is painted. A row or column performs no painting, so ensure that at least a background is painted in the cell otherwise the rectangle remains invalidated (unpainted).

Parsing the Display

After the user interacts with a cell at some point the user will edit a cell's value. One important thing to note is that the user in reality is editing the cell's FormattedValue. When committing the value the FormattedValue has to be converted back to the cell's value. This is called *parsing*. At the cell level, all of this is controlled via the DataGridViewCell:: ParseFormattedValue (int rowIndex) method.

By default, TypeConverters are used again to parse the formatted value to the real value. The DataGridView raises the CellParsing event at this time to provide you with the opportunity to modify how the cell's formatted value is parsed.

If a cell cannot correctly parse the formatted value it raises the DataError event.

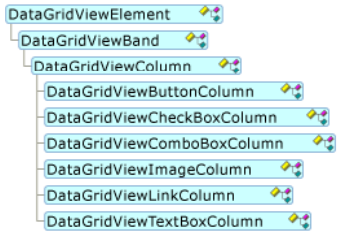
Common questions and scenarios

1) How do I prevent a particular cell from being editable?
2) How do I disable a cell?
3) How do I restrict user from setting focus to a specific cell?
4) How do I show controls in all cells regardless of edit?
5) Why does the cell text show up with "square" characters where they should be new lines?
6) How do I show icon and text in the same cell?

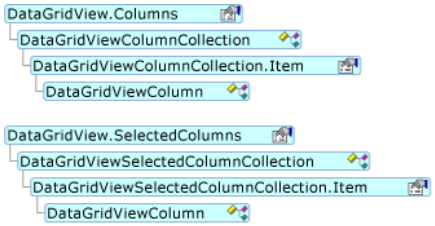
2.4 DataGridViewColumn

The schema of the DataGridView control's attached data store is expressed in the DataGridView control's columns. You can access the DataGridView control's columns by using the Columns collection. You can access the selected columns by using the SelectedColumns collection.

The DataGridViewColumn class diagram



Column Related Classes and Properties



Some of the key cell types have corresponding column types. These are derived from the DataGridViewColumn base class.

Common questions and scenarios

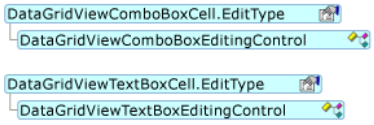
1) How do I hide a column?
2) How do I prevent the user from sorting on a column?
3) How do I sort on multiple columns?

2.5 DataGridView Editing Controls

Cells that support advanced editing functionality typically use a hosted control that is derived from a Windows Forms control. These controls also implement the IDataGridViewEditingControl interface.

The DataGridView Editing Control Class diagram

Classes that implement Editing Controls



The following editing controls are provided with the DataGridView control:

The following table illustrates the relationship among cell types, column types, and editing controls.

Cell type	Hosted control	Column type
DataGridViewButtonCell	n/a	**DataGridViewButtonColumn**
DataGridViewCheckBoxCell	n/a	**DataGridViewCheckBoxColumn**
DataGridViewComboBoxCell	DataGridViewComboBoxEditingControl	**DataGridViewComboBoxColumn**
DataGridViewImageCell	n/a	**DataGridViewImageColumn**
DataGridViewLinkCell	n/a	**DataGridViewLinkColumn**
DataGridViewTextBoxCell	DataGridViewTextBoxEditingControl	**DataGridViewTextBoxColumn**

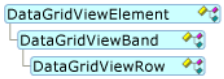
Common questions and scenarios

1) How do I hook up events on the editing control?
2) When should I remove event handlers from the editing control?
3) How do I handle the SelectedIndexChanged event?
4) How do I show controls in all cells regardless of edit?

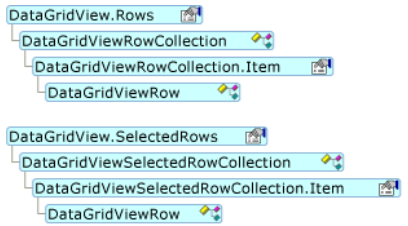
2.6 DataGridViewRow

The DataGridViewRow class displays a record's data fields from the data store to which the DataGridView control is attached. You can access the DataGridView control's rows by using the Rows collection. You can access the selected rows by using the SelectedRows collection.

The DataGridViewColumn class diagram



Row Related Classes and Properties



You can derive your own types from the DataGridViewRow class, although this will typically not be necessary. The DataGridView control has several row-related events and properties for customizing the behavior of its DataGridViewRow objects.

If you enable the DataGridView control's AllowUserToAddRows property, a special row for adding new rows appears as the last row. This row is part of the Rows collection, but it has special functionality that may require your attention. For more information, see Using the Row for New Records in the Windows Forms DataGridView Control.

Common questions and scenarios

1) How do I perform drag and drop reorder of rows?

3 Out of the box column/cell types

The DataGridView control uses several column types to display its information and enable users to modify or add information.

When you bind a DataGridView control and set the AutoGenerateColumns property to true, columns are automatically generated using default column types appropriate for the data types contained in the bound data source.

You can also create instances of any of the column classes yourself and add them to the collection returned by the Columns property. You can create these instances for use as unbound columns, or you can manually bind them. Manually bound columns are useful, for example, when you want to replace an automatically generated column of one type with a column of another type.

The following table describes the various column classes available for use in the DataGridView control:

Class	Description
DataGridViewTextBoxColumn	Used with text-based values. Generated automatically when binding to numbers and strings.
DataGridViewCheckBoxColumn	Used with Boolean and CheckState values. Generated automatically when binding to values of these types.
DataGridViewImageColumn	Used to display images. Generated automatically when binding to byte arrays, Image objects, or Icon objects.
DataGridViewButtonColumn	Used to display buttons in cells. Not automatically generated when binding. Typically used as unbound columns.
DataGridViewComboBoxColumn	Used to display drop-down lists in cells. Not automatically generated when binding. Typically data-bound manually.
DataGridViewLinkColumn	Used to display links in cells. Not automatically generated when binding. Typically data-bound manually.
Your custom column type	You can create your own column class by inheriting the DataGridViewColumn class or any of its derived classes to provide custom appearance, behavior, or hosted controls. *For more information, see How to: Customize Cells and Columns in the Windows Forms DataGridView Control by Extending Their Behavior and Appearance*

Common questions and scenarios

1) How do I hide a column?
2) How do I prevent a particular cell from being editable?
3) How do I restrict user from setting focus to a specific cell?
4) How do I make the last column wide enough to occupy all the remaining client area of the grid?

3.1 DataGridViewTextBoxColumn

The DataGridViewTextBoxColumn is a general-purpose column type for use with text-based values such as numbers and strings. In editing mode, a TextBox control is displayed in the active cell, enabling users to modify the cell value.

Cell values are automatically converted to strings for display. Values entered or modified by the user are automatically parsed to create a cell value of the appropriate data type. You can customize these conversions by handling the CellFormatting and CellParsing events of the DataGridView control.

The cell value data type of a column is specified in the ValueType property of the column.

Common questions and scenarios

1) How do I have the cell text wrap?
2) Why does the cell text show up with “square” characters where they should be new lines?
3) How do I show icon and text in the same cell?
4) How do I restrict user from setting focus to a specific cell?

3.2 DataGridViewCheckBoxColumn

The DataGridViewCheckBoxColumn is used with Boolean and CheckState values. Boolean values display as two-state or three-state check boxes, depending on the value of the ThreeState property. When the column is bound to CheckState values, the ThreeState property value is true by default.

Typically, check box cell values are intended either for storage, like any other data, or for performing bulk operations. If you want to respond immediately when users click a check box cell, you can handle the CellClick event, but this event occurs before the cell value is updated. If you need the new value at the time of the click, one option is to calculate what the expected value will be based on the current value. Another approach is to commit the change immediately, and handle the CellValueChanged event to respond to it. To commit the change when the cell is clicked, you must handle the CurrentCellDirtyStateChanged event. In the handler, if the current cell is a check box cell, call the CommitEdit method and pass in the Commit value.

3.3 DataGridViewImageColumn

The DataGridViewImageColumn is used to display images. Image columns can be populated automatically from a data source, populated manually for unbound columns, or populated dynamically in a handler for the CellFormatting event.

The automatic population of an image column from a data source works with byte arrays in a variety of image formats, including all formats supported by the Image class and the OLE Picture format used by Microsoft® Access and the Northwind sample database.

Populating an image column manually is useful when you want to provide the functionality of a DataGridViewButtonColumn, but with a customized appearance. You can handle the CellClick event to respond to clicks within an image cell.

Populating the cells of an image column in a handler for the CellFormatting event is useful when you want to provide images for calculated values or values in non-image formats. For example, you may have a "Risk" column with string values such as "high", "middle", and "low" that you want to display as icons. Alternately, you may have an "Image" column that contains the locations of images that must be loaded rather than the binary content of the images.

Common questions and scenarios

1) How do I make the image column not show any images?

3.4 DataGridViewButtonColumn

With the DataGridViewButtonColumn, you can display a column of cells that contain buttons. This is useful when you want to provide an easy way for your users to perform actions on particular records, such as placing an order or displaying child records in a separate window.

Button columns are not generated automatically when data-binding a DataGridView control. To use button columns, you must create them manually and add them to the collection returned by the Columns property.

You can respond to user clicks in button cells by handling the CellClick event.

3.5 DataGridViewComboBoxColumn

With the DataGridViewComboBoxColumn, you can display a column of cells that contain drop-down list boxes. This is useful for data entry in fields that can only contain particular values, such as the Category column of the Products table in the Northwind sample database.

You can populate the drop-down list used for all cells the same way you would populate a ComboBox drop-down list, either manually through the collection returned by the Items property, or by binding it to a data source through the DataSource, DisplayMember, and ValueMember properties. For more information, see ComboBox Control (Windows Forms).

You can bind the actual cell values to the data source used by the DataGridView control by setting the DataPropertyName property of the DataGridViewComboBoxColumn.

Combo box columns are not generated automatically when data-binding a DataGridView control. To use combo box columns, you must create them manually and add them to the collection returned by the Columns property. Alternatively you can use the designer and change a column type to a combo box column and set properties accordingly.

3.5.1 DataErrors and the combo box column

Sometimes when working the DataGridViewComboBoxColumn and modifying either the cell's value or the combo box items collection, sometimes the DataError event might be raised. This is by design because of the data validation that the combo box cell performs. When the combo box cell attempts to draw its content it has to go from the cell value to the formatted value. This conversion requires looking up the value in the combo box items and getting the display value for the item. During this process, if the cell's value cannot be found in the combo box items collection it raises the DataError event. Ignoring the DataError event might keep the cell from displaying the correct formatted value.

Common questions and scenarios

1) How do I enable typing in the combo box cell?
2) How do I handle the SelectedIndexChanged event?
3) How do I have a combo box column display a sub set of data based upon the value of a different combo box column?

3.6 DataGridViewLinkColumn

With the DataGridViewLinkColumn, you can display a column of cells that contain hyperlinks. This is useful for URL values in the data source or as an alternative to the button column for special behaviors such as opening a window with child records.

Link columns are not generated automatically when data-binding a DataGridView control. To use link columns, you must create them manually and add them to the collection returned by the Columns property.

You can respond to user clicks on links by handling the CellContentClick event. This event is distinct from the CellClick and CellMouseClick events, which occur when a user clicks anywhere in a cell.

The DataGridViewLinkColumn class provides several properties for modifying the appearance of links before, during, and after they are clicked.

4 Working with Data

In most cases you'll be using the DataGridView with data. There are many tasks that you might need to do when working with your data in the DataGridView. You might need to validate data that the user entered or maybe you might need to format the data. The DataGridView can display data in three distinct modes: bound, unbound, and virtual. Each mode has its own features and reasons to choose it over the other. Regardless of databound mode it is common for the grid to raise the DataError event when something goes wrong when working with data. Understanding why this event occurs will make the event more helpful.

4.1 Data Entry and Validation Events

When the user enters data – either at the row or cell level you might want to validate the data and maybe inform the user about invalid data. Like normal Windows Forms, there are Validating and Validated events at the cell and row level. Validating events can be canceled. There are also Enter and Leave events for when the user moves between cells or rows. Lastly there are events for when the user starts editing a cell. Understand the order of all these events will be helpful.

4.1.1 Order of Validation Events

The following identifies the order of validation, enter/leave and begin/end edit events. The EditMode is EditOnEnter.

When moving from cell to cell (in the same row)

1) Cell Leave (old cell)
2) Cell Validating/ed (old cell)
3) Cell EndEdit (old cell)
4) Cell Enter (new cell)
5) Cell BeginEdit (new cell)

When moving from one row to another you get:

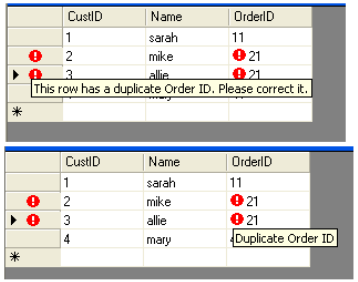
1) Cell Leave (old cell), Row leave (old row)
2) Cell Validating/ed (old cell)
3) Cell EndEdit (old cell)
4) Row Validating/ed (old row)
5) Row Enter (new row)
6) Cell Enter (new cell)
7) Cell BeginEdit (new cell)

4.1.2 Validating Data

When validating user data, you will usually validate the data at the cell level when the grid is not databound, and at the row level when the grid is databound. Sometimes when databound you will perform validation both at the cell and row level.

4.1.2.1 Displaying error information

When you do encounter data that is invalid you usually notify the user about it. There are many ways to do this; the conventional way is to use a message box. The DataGridview has the ability to show an error icon for the cell and rows to indicate that invalid data was entered. The error icon has a tooltip associated with it to provide informational about what is in error:



Common questions and scenarios

1) How do I show the error icon when the user is editing the cell?

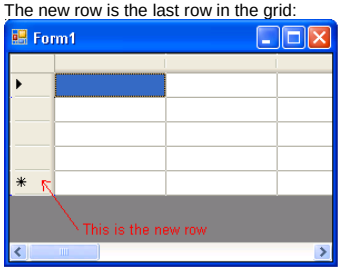
4.1.3 Data Entry in the New Row

When you use a DataGridView for editing data in your application, you will often want to give your users the ability to add new rows of data to the data store. The DataGridView control supports this functionality by providing a row for new records, which is always shown as the last row. It is marked with an asterisk (*) symbol in its row header. The following sections discuss some of the things you should consider when you program with the row for new records enabled.

4.1.3.1 Displaying the Row for New Records

Use the AllowUserToAddRows property to indicate whether the row for new records is displayed. The default value of this property is true.

The new row is the last row in the grid:



For the data bound case, the row for new records will be shown if the AllowUserToAddRows property of the control and the IBindingList.AllowNew property of the data source are both true. If either is false then the row will not be shown.

4.1.3.2 Populating the Row for New Records with Default Data

When the user selects the row for new records as the current row, the DataGridView control raises the DefaultValuesNeeded event. This event provides access to the new DataGridViewRow and enables you to populate the new row with default data.

The following code example demonstrates how to specify default values for new rows using the DefaultValuesNeeded event.

```
private void dataGridView1_DefaultValuesNeeded(object sender,
                                DataGridViewRowEventArgs e)
{
    e.Row.Cells["Region"].Value = "WA";
    e.Row.Cells["City"].Value = "Redmond";
    e.Row.Cells["PostalCode"].Value = "98052-6399";
    e.Row.Cells["Region"].Value = "NA";
    e.Row.Cells["Country"].Value = "USA";
    e.Row.Cells["CustomerID"].Value = NewCustomerId();
}
```

4.1.3.3 The Rows Collection and the New Row

The row for new records is contained in the DataGridView control's Rows collection, so the following line returns the new row:

```
DataGridViewRow row = dataGridView1.Rows[dataGridView1.Rows.Count - 1];
```

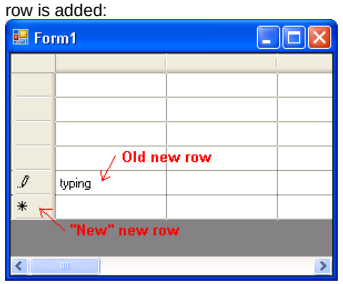
Even though the new row is in the rows collection it does behaves differently in two respects:

- The row for new records cannot be removed from the Rows collection programmatically. An InvalidOperationException is thrown if this is attempted. The user also cannot delete the row for new records. The DataGridViewRowCollection.Clear method does not remove this row from the Rows collection.

- No row can be added after the row for new records. An InvalidOperationException is raised if this is attempted. As a result, the row for new records is always the last row in the DataGridView control. The methods on DataGridViewRowCollection that add rows—Add, AddCopy, and AddCopies—all call insertion methods internally when the row for new records is present.

4.1.3.4 Typing in the New Row

Before a user starts to type in the new row the row's IsNewRow property returns true. When the user starts to type in the new row, that row no longer is considered the new row, but a "new" new row is added:



When this "new" new row is added the UserAddedRow event fires with the Row event args property identifying the "new" new row. If the user hits the Escape key at this stage the "new" new row is removed. This causes the UserDeletingRow event to fire with the Row event args property identifying the "new" new row.

4.1.3.5 Visual Customization of the Row for New Records

When the row for new records is created, it is based on the row specified by the RowTemplate property. Any cell styles that are not specified for this row are inherited from other properties. For more information about cell style inheritance, see the Cell Styles topic later in this document.

The initial values displayed by cells in the row for new records are retrieved from each cell's DefaultNewRowValue property. For cells of type DataGridViewImageCell, this property returns a placeholder image. Otherwise, this property returns null. You can override this property to return a custom value. However, these initial values can be replaced by a DefaultValuesNeeded event handler when focus enters the row for new records.

The standard icons for this row's header, which are an arrow or an asterisk, are not exposed publicly. If you want to customize the icons, you will need to create a custom DataGridViewRowHeaderCell class.

The standard icons use the ForeColor property of the DataGridViewCellStyle in use by the row header cell. The standard icons are not rendered if there is not enough space to display them completely.

If the row header cell has a string value set, and if there is not enough room for both the text and icon, the icon is dropped first.

4.1.3.6 Sorting the New Row

In unbound mode, new records will always be added to the end of the DataGridView even if the user has sorted the content of the DataGridView. The user will need to apply the sort again in order to sort the row to the correct position; this behavior is similar to that of the ListView control.

In data bound and virtual modes, the insertion behavior when a sort is applied will be dependent on the implementation of the data model. For ADO.NET, the row is immediately sorted into the correct position.

4.1.3.7 Other Notes on the Row for New Records

You cannot set the Visible property of this row to false. An InvalidOperationException is raised if this is attempted.

The row for new records is always created in the unselected state.

4.1.3.8 Virtual Mode

If you are implementing virtual mode, you will need to track when a row for new records is needed in the data model and when to roll back the addition of the row. The exact implementation of this functionality depends on the implementation of the data model and its transaction semantics, for example, whether commit scope is at the cell or row level. See the Virtual Mode topic later in this document for more info.

4.2 Working with Null values

When working with datasource such as a database or a business object it is common to deal with null values. Null values can be an actual null (nothing in VB) or a database "null value" (DBNull.Value). When working with these values you'll need to decide how you will display them. In addition there are valid reasons for when you'll want to write a null value. Using the cell style's NullValue and DataSourceNullValue properties you can change how the DataGridView works with null values.

4.2.1 NullValue

The DataGridViewCellStyle.NullValue property should have been called FormattedNullValue, but it was too late to make this change. Hopefully this provides a clue into how the NullValue is used – at formatting time. When a cell's value is "null" (equal to Null or DBNull.Value) the value in the DataGridViewCellStyle.NullValue property is used for display. The default value of this property is based upon the column:

DataGridView column	Column's DefaultCellStyle.NullValue
TextBoxColumn	String.Empty ("")
ImageColumn	Null image ( )
ComboBoxColumn	String.Empty ("")
ButtonColumn	String.Empty ("")
LinkColumn	String.Empty ("")
CheckBoxColumn	Default is based upon the ThreeState property. If true default value is CheckState.Indeterminate; otherwise it is false (unchecked).

What is important to understand is that the NullValue is also used when the user enters data. For example, if the user enters String.Empty into a text box cell then Null is entered as the cell's value (check out the DataSourceNullValue below for more details on what is actually entered as the cell's value).

4.2.2 DataSourceNullValue

The DataGridViewCellStyle.DataSourceNullValue property could have been called ParseNullValue if the other property was FormattedNullValue, but in the end DataSourceNullValue made good sense. The DataSourceNullValue property is used when writing the value of "Null" to the cell's value. In databound scenarios this value gets written to the database or business object. This is important to control as business objects and databases have different concepts of null. Usually you'll want to set DataSourceNullValue to null when working with business objects and DBNull.Value when working with databases. The default value of DataSourceNullValue is DBNull.Value.

4.3 DataError event

The DataError event gets its own topic because it is quite common for the DataError event to occur when working with data and the DataGridView. Basically the DataError event occurs anytime the grid is working with data and the grid cannot read/write or convert the cell's data or when an exception occurs when attempting to perform certain edit operations.

DataError Event for Edit Operations
The following list identifies edit operations when the DataError event might be raised if an exception occurs:

- Canceling an edit
- Ending an edit
- Committing an edit
- Deleting a row
- Refreshing an edit (via the RefreshEdit method)
- When we attempt to push a cell's value to the datasource
- Initializing the editing control's\cell's value (via setting the cell's FormattedValue property or cell's InitializeEditingControl method.)

DataError Contexts
The following list identifies different DataErrorContexts and provides more detail into when a certain context might occur

DataErrorContext	When it might occur
Formatting	When attempting to retrieve the cell's formatted value.
Display	When attempting to paint the cell or calculate the cell's tooltiptext. Note that these operations usually also require getting the cell's formatted value, so the error context is OR'd together.
PreferredSize	When calculating the preferred size of a cell. This usually also requires getting the cell's formatted value also.
RowDeletion	Any exception raised when deleting a row.
Parsing	When exceptions occur when committing, ending or canceling an edit. Usually OR'd in with other error contexts
Commit	When exceptions occur when committing an edit. Usually OR'd with other error contexts
InitialValueRestoration	When exceptions occur while either initializing the editing control/cell's value, or Canceling an edit
LeaveControl	When exceptions occur while attempting to validate grid data when the grid is losing focus. Usually OR'd with other error contexts.
CurrentCellChange	When exceptions occur while validating\updating\committing\getting cell content when the current cell changes. Usually OR'd with other error contexts.
Scroll	When exceptions occur while validating\updating\committing\getting cell content when the current cell changes as a result of scrolling.
ClipboardContent	When exceptions occur while attempting to get the formatted value of a cell while creating the clipboard content.

4.4 Databound modes

4.4.1 Unbound

Unbound mode is suitable for displaying relatively small amounts of data that you manage programmatically. You do not attach the DataGridView control directly to a data source as in bound mode. Instead, you must populate the control yourself, typically by using the DataGridViewRowCollection.Add method.

Unbound mode can be particularly useful for static, read-only data, or when you want to provide your own code that interacts with an external data store. When you want your users to interact with an external data source, however, you will typically use bound mode.

4.4.2 Bound

Bound mode is suitable for managing data using automatic interaction with the data store. You can attach the DataGridView control directly to its data source by setting the DataSource property. When the control is data bound, data rows are pushed and pulled without the need of

explicit management on your part. When the AutoGenerateColumns property is true, each column in your data source will cause a corresponding column to be created in the control. If you prefer to create your own columns, you can set this property to false and use the DataPropertyName property to bind each column when you configure it. This is useful when you want to use a column type other than the types that are generated by default. For more info on databinding the DataGridView, check out the following MSDN article: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnforms/html/winforms11162004.asp Also, check out the DataBinding FAQ posted at WindowsForms.net

4.4.2.1 Valid Data Sources

Binding data to the DataGridView control is straightforward and intuitive, and in many cases it is as simple as setting the DataSource property. When you bind to a data source that contains multiple lists or tables, you need to set the DataMember property to a string to specify the list or table to bind to.

The DataGridView control supports the standard Windows Forms data binding model, so it will bind to instances of classes described in the following list:

- Any class that implements the IList interface, including one-dimensional arrays.
- Any class that implements the IListSource interface, such as the DataTable and DataSet classes.
- Any class that implements the IBindingList interface, such as the BindingList class.
- Any class that implements the IBindingListView interface, such as the BindingSource class.

List Change Notification

One of the most important parts when databinding is for the list to support change notifications. This is only important if you want the DataGridView to be updated when the list changes such as add, updates and deletes. Only datasources that implement the IBindingList support change notification. Lists such as arrays or collections do not support change notification by default.

The BindingSource component is the preferred data source because it can bind to a wide variety of data sources and can resolve many data binding issues automatically. Typically, you will bind to a BindingSource component and bind the BindingSource component to another data source or populate it with business objects. The BindingList<T> class can also be used to create a custom list based upon a type.

Object Change Notification

Once you have a datasource the objects in the datasource optionally can implement change notification for public properties. This is done by either providing a “PropertyName”Changed event for the property or by implementing the INotifyPropertyChanged interface. The INotifyPropertyChanged interface is new in VS 2005 and can be used with BindingList<T> to create bindable lists. When your datasource is a BindingSource, the objects in the list do not need to implement change notification.

4.4.3 Virtual

With virtual mode, you can implement your own data management operations. This is necessary to maintain the values of unbound columns in bound mode when the control is sorted by bound columns. The primary use of virtual mode, however, is to optimize performance when interacting with large amounts of data.

You attach the DataGridView control to a cache that you manage, and your code controls when data rows are pushed and pulled. To keep the memory footprint small, the cache should be

similar in size to the number of rows currently displayed. When the user scrolls new rows into view, your code requests new data from the cache and optionally flushes old data from memory.

When you are implementing virtual mode, you will need to track when a new row is needed in the data model and when to rollback the addition of the new row. The exact implementation of this functionality will depend on the implementation of the data model and the transaction semantics of the data model; whether commit scope is at the cell or row level.

For more info about Virtual mode, see the “Virtual Mode” topic later in this document.

4.4.4 Mixed mode – Bound and Unbound

The data you display in the DataGridView control will normally come from a data source of some kind, but you might want to display a column of data that does not come from the data source. This kind of column is called an unbound column.

You can add unbound columns to a DataGridView control in bound mode. This is useful when you want to display a column of buttons or links that enable users to perform actions on specific rows. It is also useful to display columns with values calculated from bound columns. You can populate the cell values for calculated columns in a handler for the CellFormatting event. If you are using a DataSet or DataTable as the data source, however, you might want to use the DataColumn.Expression property to create a calculated column instead. In this case, the DataGridView control will treat calculated column just like any other column in the data source.

Sorting by unbound columns in bound mode is not supported. If you create an unbound column in bound mode that contains user-editable values, you must implement virtual mode to maintain these values when the control is sorted by a bound column.

Virtual mode should also be used in mix mode when the additional data being displayed cannot be calculated based upon the data that the grid is bound to or when the data is updated frequently. For more info about Virtual mode, see the “Virtual Mode” topic later in this document.

Common questions and scenarios

5 Overview of features

5.1 Styling

The DataGridView control makes it easy to define the basic appearance of cells and the display formatting of cell values. You can define appearance and formatting styles for individual cells, for cells in specific columns and rows, or for all cells in the control by setting the properties of the DataGridViewCellStyle objects accessed through various DataGridView control properties. Additionally, you can modify these styles dynamically based on factors such as the cell value by handling the CellFormatting event.

Each cell within the DataGridView control can have its own style, such as text format, background color, foreground color, and font. Typically, however, multiple cells will share particular style characteristics.

Groups of cells that share styles may include all cells within particular rows or columns, all cells that contain particular values, or all cells in the control. Because these groups overlap, each cell may get its styling information from more than one place. For example, you may want every cell in a DataGridView control to use the same font, but only cells in currency columns to use currency format, and only currency cells with negative numbers to use a red foreground color.

5.1.1 The DataGridViewCellStyle Class

The DataGridViewCellStyle class contains the following properties related to visual style:

BackColor and ForeColor, SelectionBackColor and SelectionForeColor, Font

This class also contains the following properties related to formatting:

Format and FormatProvider, NullValue and DataSourceNullValue, WrapMode, Alignment, Padding

5.1.2 Using DataGridViewCellStyle Objects

You can retrieve DataGridViewCellStyle objects from various properties of the DataGridView, DataGridViewColumn, DataGridViewRow, and DataGridViewCell classes and their derived classes. If one of these properties has not yet been set, retrieving its value will create a new DataGridViewCellStyle object. You can also instantiate your own DataGridViewCellStyle objects and assign them to these properties.

You can avoid unnecessary duplication of style information by sharing DataGridViewCellStyle objects among multiple DataGridView elements. Because the styles set at the control, column, and row levels filter down through each level to the cell level, you can also avoid style duplication by setting only those style properties at each level that differ from the levels above. This is described in more detail in the Style Inheritance section that follows.

The following table lists the primary properties that get or set DataGridViewCellStyle objects.

Property	Classes	Description
DefaultCellStyle	DataGridView, DataGridViewColumn, DataGridViewRow, and derived classes	Gets or sets default styles used by all cells in the entire control (including header cells), in a column, or in a row.
RowsDefaultCellStyle	DataGridView	Gets or sets default cell styles used by all rows in the control. This does not include header cells.

AlternatingRowsDefaultCellStyle	DataGridView	Gets or sets default cell styles used by alternating rows in the control. Used to create a ledger-like effect.
RowHeadersDefaultCellStyle	DataGridView	Gets or sets default cell styles used by the control's row headers. Overridden by the current theme if visual styles are enabled.
ColumnHeadersDefaultCellStyle	DataGridView	Gets or sets default cell styles used by the control's column headers. Overridden by the current theme if visual styles are enabled.
Style	DataGridViewCell and derived classes	Gets or sets styles specified at the cell level. These styles override those inherited from higher levels.
InheritedStyle	DataGridViewCell, DataGridViewRow, DataGridViewColumn, and derived classes	Gets all the styles currently applied to the cell, row, or column, including styles inherited from higher levels.

As mentioned above, getting the value of a style property automatically instantiates a new DataGridViewCellStyle object if the property has not been previously set. To avoid creating these objects unnecessarily, the row and column classes have a HasDefaultCellStyle property that you can check to determine whether the DefaultCellStyle property has been set. Similarly, the cell classes have a HasStyle property that indicates whether the Style property has been set.

Each of the style properties has a corresponding PropertyNameChanged event on the DataGridView control. For row, column, and cell properties, the name of the event begins with "Row", "Column", or "Cell" (for example, RowDefaultCellStyleChanged). Each of these events occurs when the corresponding style property is set to a different DataGridViewCellStyle object. These events do not occur when you retrieve a DataGridViewCellStyle object from a style property and modify its property values. To respond to changes to the cell style objects themselves, handle the CellStyleContentChanged event.

5.1.3 Style Inheritance

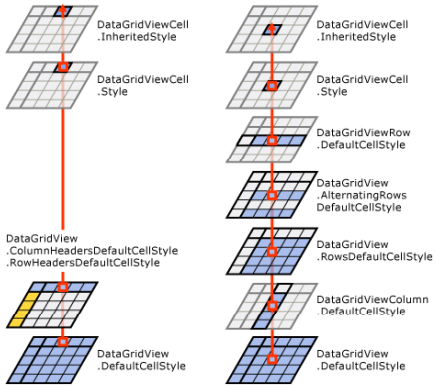
Each DataGridViewCell gets its appearance from its InheritedStyle property. The DataGridViewCellStyle object returned by this property inherits its values from a hierarchy of properties of type DataGridViewCellStyle. These properties are listed below in the order in which the InheritedStyle for non-header cells obtains its values.

1. DataGridViewCell.Style
2. DataGridViewRow.DefaultCellStyle
3. AlternatingRowsDefaultCellStyle (only for cells in rows with odd index numbers)
4. RowsDefaultCellStyle
5. DataGridViewColumn.DefaultCellStyle
6. DefaultCellStyle

For row and column header cells, the InheritedStyle property is populated by values from the following list of source properties in the given order.

1. DataGridViewCell.Style
2. ColumnHeadersDefaultCellStyle or RowHeadersDefaultCellStyle
3. DefaultCellStyle

The following diagram illustrates this process.



You can also access the styles inherited by specific rows and columns. The column InheritedStyle property inherits its values from the following properties.

1. DataGridViewColumn.DefaultCellStyle
2. DefaultCellStyle

The row InheritedStyle property inherits its values from the following properties.

1. DataGridViewRow.DefaultCellStyle
2. AlternatingRowsDefaultCellStyle (only for cells in rows with odd index numbers)
3. RowsDefaultCellStyle
4. DefaultCellStyle

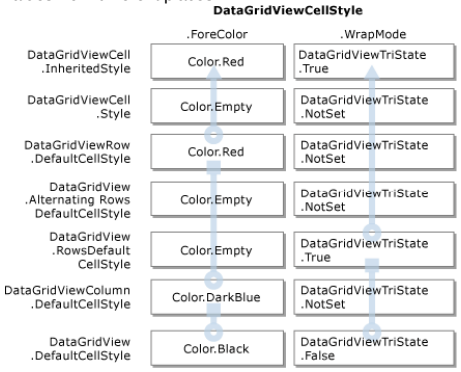
For each property in a DataGridViewCellStyle object returned by an InheritedStyle property, the property value is obtained from the first cell style in the appropriate list that has the corresponding property set to a value other than the DataGridViewCellStyle class defaults.

The following table illustrates how the ForeColor property value for an example cell is inherited from its containing column.

Property of type DataGridViewCellStyle	Example ForeColor value for retrieved object
DataGridViewCell.Style	Color.Empty
DataGridViewRow.DefaultCellStyle	Color.Red
AlternatingRowsDefaultCellStyle	Color.Empty
RowsDefaultCellStyle	Color.Empty
DataGridViewColumn.DefaultCellStyle	Color.DarkBlue
DefaultCellStyle	Color.Black

In this case, the System.Drawing.Color.Red value from the cell's row is the first real value on the list. This becomes the ForeColor property value of the cell's InheritedStyle.

The following diagram illustrates how different DataGridViewCellStyle properties can inherit their values from different places.



By taking advantage of style inheritance, you can provide appropriate styles for the entire control without having to specify the same information in multiple places.

Although header cells participate in style inheritance as described, the objects returned by the ColumnHeadersDefaultCellStyle and RowHeadersDefaultCellStyle properties of the DataGridView control have initial property values that override the property values of the object returned by the DefaultCellStyle property. If you want the properties set for the object returned by the DefaultCellStyle property to apply to row and column headers, you must set the corresponding properties of the objects returned by the ColumnHeadersDefaultCellStyle and RowHeadersDefaultCellStyle properties to the defaults indicated for the DataGridViewCellStyle class.

> **Note:** If visual styles are enabled, the row and column headers (except for the TopLeftHeaderCell) are automatically styled by the current theme, overriding any styles specified by these properties. Set the EnableHeadersVisualStyle property to false if you want headers to not use XP's visual styles.

The DataGridViewButtonColumn, DataGridViewImageColumn, and DataGridViewCheckBoxColumn types also initialize some values of the object returned by the column DefaultCellStyle property. For more information, see the reference documentation for these types.

5.1.4 Setting Styles Dynamically

To customize the styles of cells with particular values, implement a handler for the CellFormatting event. Handlers for this event receive an argument of the DataGridViewCellFormattingEventArgs type. This object contains properties that let you determine the value of the cell being formatted along with its location in the DataGridView control. This object also contains a CellStyle property that is initialized to the value of the InheritedStyle property of the cell being formatted. You can modify the cell style properties to specify style information appropriate to the cell value and location.

> **Note:** The RowPrePaint and RowPostPaint events also receive a DataGridViewCellStyle object in the event data, but in their case, it is a copy of the row InheritedStyle property for read-only purposes, and changes to it do not affect the control.

You can also dynamically modify the styles of individual cells in response to events such as the CellMouseEnter and CellMouseLeave events. For example, in a handler for the CellMouseEnter event, you could store the current value of the cell background color (retrieved through the cell's Style property), then set it to a new color that will highlight the cell when the mouse hovers over it. In a handler for the CellMouseLeave event, you can then restore the background color to the original value.

> **Note:** Caching the values stored in the cell's Style property is important regardless of whether a particular style value is set. If you temporarily replace a style setting, restoring it to its original "not set" state ensures that the cell will go back to inheriting the style setting from a higher level. If you need to determine the actual style in effect for a cell regardless of whether the style is inherited, use the cell's InheritedStyle property.

5.2 Custom painting

The DataGridView control provides several properties that you can use to adjust the appearance and basic behavior (look and feel) of its cells, rows, and columns. If you have requirements that go beyond the capabilities of the DataGridViewCellStyle class, you can perform custom drawing of the cell or row content. To paint cells and rows yourself, you can handle various DataGridView painting events such as RowPrePaint, CellPainting and RowPostPaint.

5.2.1 Paint Parts

One important part of custom painting is the concept of paint parts. The DataGridViewPainParts enumeration is used to specify what parts a cell paints. Enum values can be combined together to have a cell paint or not paint specific parts. Here are the different parts:

PaintPart	Example ForeColor value for retrieved object
All	All parts are painted
Background	The background of the cell is painted using the cell's background color (1)
Border	The borders are painted
ContentBackground	The background part of the cell's content is painted. (2)
ContentForeground	The foreground part of the cell's content is painted (2)
ErrorIcon	The error icon is painted
Focus	The focus rectangle for the cell is painted
None	No parts are painted (1)
SelectionBackground	The background is painted selected if the cell is selected.

Notes

1) If a cell does not paint its background then nothing is painted. A row or column performs no painting, so ensure that at least the cell's background is painted or you perform your own custom background painting; otherwise the rectangle remains invalidated (unpainted).
2) Each cell determines what it paints as content foreground and content background as described by the following list:

Cell Type	Content Foreground	Content Background
Text box	Cell text is painted	Nothing painted

Button	Cell text is painted	Button is painted
Combo box	Cell text is painted	Combo box is painted
Check box	Check box is painted	Nothing painted
Link	Cell text is painted as a link	Nothing is painted
Image	Cell image is painted	Nothing painted
Column Header	Column header text	Sort Glyph is painted
Row Header	Row header text	Current row triangle, edit pencil and new row indicator is painted

5.2.2 Row Pre Paint and Post Paint

You can control the appearance of DataGridView rows by handling one or both of the DataGridView.RowPrePaint and DataGridView.RowPostPaint events. These events are designed so that you can paint only what you want to while letting the DataGridView control paint the rest. For example, if you want to paint a custom background, you can handle the DataGridView.RowPrePaint event and let the individual cells paint their own foreground content. In the RowPrePaint event you can set the PaintParts event args property to easily customize how the cells paint. For example, if you want to keep cells from painting any selection or focus, your RowPrePaint event would set the PaintParts property like so:

```
e.PaintParts = DataGridViewPaintParts.All &
      ~(DataGridViewPaintParts.Focus |
        DataGridViewPaintParts.SelectionBackground);
```

Which could also be written as:

```
e.PaintParts = (DataGridViewPaintParts.Background |
                DataGridViewPaintParts.Border |
                DataGridViewPaintParts.ContentBackground |
                DataGridViewPaintParts.ContentForeground |
                DataGridViewPaintParts.ErrorIcon);
```

Alternately, you can let the cells paint themselves and add custom foreground content in a handler for the DataGridView.RowPostPaint event. You can also disable cell painting and paint everything yourself in a DataGridView.RowPrePaint event handler

5.3 Autosizing

The DataGridView control provides numerous options for customizing the sizing behavior of its columns and rows. Typically, DataGridView cells do not resize based on their contents. Instead, they clip any display value that is larger than the cell. If the content can be displayed as a string, the cell displays it in a ToolTip.

By default, users can drag row, column, and header dividers with the mouse to show more information. Users can also double-click a divider to automatically resize the associated row, column, or header band based on its contents. Columns share the available width of the control by default, so if users can resize the control—for example, if it is docked to a resizable form—they can also change the available display space for all columns.

The DataGridView control provides properties, methods, and events that enable you to customize or disable all of these user-directed behaviors. Additionally, you can programmatically resize rows, columns, and headers to fit their contents, or you can configure them to automatically resize themselves whenever their contents change.

Common questions and scenarios

1) How do I make the last column wide enough to occupy all the remaining client area of the grid?

5.3.1 Sizing Options in the Windows Forms DataGridView Control

DataGridView rows, columns, and headers can change size as a result of many different occurrences. The following table shows these occurrences.

Occurrence	Description
User resize	Users can make size adjustments by dragging or double-clicking row, column, or header dividers.
Control resize	In column fill mode, column widths change when the control width changes; for example, when the control is docked to its parent form and the user resizes the form.
Cell value change	In content-based automatic sizing modes, sizes change to fit new display values.
Method call	Programmatic content-based resizing lets you make opportunistic size adjustments based on cell values at the time of the method call.
Property setting	You can also set specific height and width values.

By default, user resizing is enabled, automatic sizing is disabled, and cell values that are wider than their columns are clipped.

The following table shows scenarios that you can use to adjust the default behavior or to use specific sizing options to achieve particular effects.

Scenario	Implementation
Use column fill mode for displaying similarly sized data in a relatively small number of columns that occupy the entire width of the control without displaying the horizontal scroll bar.	Set the AutoSizeColumnsMode property to Fill.
Use column fill mode with display values of varying sizes.	Set the **AutoSizeColumnsMode** property to **Fill**. Initialize relative column widths by setting the column FillWeight properties or by calling the control AutoResizeColumns method after filling the control with data.
Use column fill mode with values of varying importance.	Set the **AutoSizeColumnsMode** property to **Fill**. Set large MinimumWidth values for columns that must always display some of their data or use a sizing option other than fill mode for specific columns.
Use column fill mode to avoid displaying the control background.	Set the AutoSizeMode property of the last column to Fill and use other sizing options for the other columns.
Display a fixed-width column, such as an icon or ID column.	Set **AutoSizeMode** to None and Resizable to False for the column. Initialize its width by setting the Width property or by calling the control AutoResizeColumn method after filling the control with data.
Adjust sizes automatically whenever cell contents change to avoid clipping and to optimize the use of space.	Set an automatic sizing property to a value that represents a content-based sizing mode. To avoid a performance penalty when working with large amounts of data, use a sizing mode that calculates displayed rows only.
Adjust sizes to fit values in displayed rows to avoid performance penalties when working with many rows.	Use the appropriate sizing-mode enumeration values with automatic or programmatic resizing. To adjust sizes to fit values in newly displayed rows while scrolling, call a resizing method in a Scroll event handler. To customize user double-click resizing so that only values in displayed rows determine the new sizes, call a resizing method in a RowDividerDoubleClick or ColumnDividerDoubleClick event handler.
Adjust sizes to fit cell contents only at specific times to avoid performance penalties or to enable user resizing.	Call a content-based resizing method in an event handler. For example, use the DataBindingComplete event to initialize sizes after binding, and handle the CellValidated or CellValueChanged event to adjust sizes to compensate for user edits or changes in a bound data source.
Adjust row heights for multiline cell contents.	Ensure that column widths are appropriate for displaying paragraphs of text and use automatic or programmatic content-based row sizing to adjust the heights. Also ensure that cells with multiline content are displayed using a WrapMode cell style value of True. Typically, you will use an automatic column sizing mode to maintain column widths or set them to specific widths before row heights are adjusted.

5.3.2 Resizing with the Mouse

By default, users can resize rows, columns, and headers that do not use an automatic sizing mode based on cell values. To prevent users from resizing with other modes, such as column fill mode, set one or more of the following DataGridView properties:

- AllowUserToResizeColumns
- AllowUserToResizeRows
- ColumnHeadersHeightSizeMode
- RowHeadersWidthSizeMode

You can also prevent users from resizing individual rows or columns by setting their Resizable properties. By default, the Resizable property value is based on the AllowUserToResizeColumns property value for columns and the AllowUserToResizeRows property value for rows. If you explicitly set Resizable to True or False, however, the specified value overrides the control value is for that row or column. Set Resizable to NotSet to restore the inheritance.

Because NotSet restores the value inheritance, the Resizable property will never return a NotSet value unless the row or column has not been added to a DataGridView control. If you need to determine whether the Resizable property value of a row or column is inherited, examine its State property. If the State value includes the ResizableSet flag, the Resizable property value is not inherited.

5.3.3 Automatic Sizing

There are two kinds of automatic sizing in the DataGridView control: column fill mode and content-based automatic sizing.

Column fill mode causes the visible columns in the control to fill the width of the control's display area. For more information about this mode, see the Column Fill Mode section below.

You can also configure rows, columns, and headers to automatically adjust their sizes to fit their cell contents. In this case, size adjustment occurs whenever cell contents change.

> **Note:** If you maintain cell values in a custom data cache using virtual mode, automatic sizing occurs when the user edits a cell value but does not occur when you alter a cached value outside of a CellValuePushed event handler. In this case, call the UpdateCellValue method to force the control to update the cell display and apply the current automatic sizing modes.

If content-based automatic sizing is enabled for one dimension only—that is, for rows but not columns, or for columns but not rows—and WrapMode is also enabled, size adjustment also occurs whenever the other dimension changes. For example, if rows but not columns are configured for automatic sizing and WrapMode is enabled, users can drag column dividers to change the width of a column and row heights will automatically adjust so that cell contents are still fully displayed.

If you configure both rows and columns for content-based automatic sizing and WrapMode is enabled, the DataGridView control will adjust sizes whenever cell contents changed and will use an ideal cell height-to-width ratio when calculating new sizes.

To configure the sizing mode for headers and rows and for columns that do not override the control value, set one or more of the following DataGridView properties:

- ColumnHeadersHeightSizeMode
- RowHeadersWidthSizeMode
- AutoSizeColumnsMode
- AutoSizeRowsMode

To override the control's column sizing mode for an individual column, set its AutoSizeMode property to a value other than NotSet. The sizing mode for a column is actually determined by its InheritedAutoSizeMode property. The value of this property is based on the column's AutoSizeMode property value unless that value is NotSet, in which case the control's AutoSizeColumnsMode value is inherited.

Use content-based automatic resizing with caution when working with large amounts of data. To avoid performance penalties, use the automatic sizing modes that calculate sizes based only on the displayed rows rather than analyzing every row in the control. For maximum performance, use programmatic resizing instead so that you can resize at specific times, such as immediately after new data is loaded.

Content-based automatic sizing modes do not affect rows, columns, or headers that you have hidden by setting the row or column Visible property or the control RowHeadersVisible or ColumnHeadersVisible properties to false. For example, if a column is hidden after it is automatically sized to fit a large cell value, the hidden column will not change its size if the row containing the large cell value is deleted. Automatic sizing does not occur when visibility changes, so changing the column Visible property back to true will not force it to recalculate its size based on its current contents.

Programmatic content-based resizing affects rows, columns, and headers regardless of their visibility.

5.3.4 Programmatic Resizing

When automatic sizing is disabled, you can programmatically set the exact width or height of rows, columns, or headers through the following properties:

- RowHeadersWidth
- ColumnHeadersHeight
- DataGridViewRow.Height
- DataGridViewColumn.Width

You can also programmatically resize rows, columns, and headers to fit their contents using the following methods:

- AutoResizeColumn
- AutoResizeColumns
- AutoResizeColumnHeadersHeight
- AutoResizeRow
- AutoResizeRows
- AutoResizeRowHeadersWidth

These methods will resize rows, columns, or headers once rather than configuring them for continuous resizing. The new sizes are automatically calculated to display all cell contents without clipping. When you programmatically resize columns that have InheritedAutoSizeMode property values of Fill, however, the calculated content-based widths are used to proportionally adjust the column FillWeight property values, and the actually column widths are then calculated according to these new proportions so that all columns fill the available display area of the control.

Programmatic resizing is useful to avoid performance penalties with continuous resizing. It is also useful to provide initial sizes for user-resizable rows, columns, and headers, and for column fill mode.

You will typically call the programmatic resizing methods at specific times. For example, you might programmatically resize all columns immediately after loading data, or you might programmatically resize a specific row after a particular cell value has been modified.

5.3.5 Customizing Content-based Sizing Behavior

You can customize sizing behaviors when working with derived DataGridView cell, row, and column types by overriding the DataGridViewCell.GetPreferredSize(), DataGridViewRow.GetPreferredHeight(), or DataGridViewColumn.GetPreferredWidth() methods or by calling protected resizing method overloads in a derived DataGridView control. The protected resizing method overloads are designed to work in pairs to achieve an ideal cell height-to-width ratio, avoiding overly wide or tall cells. For example, if you call the AutoResizeRows(DataGridViewAutoSizeRowsMode,Boolean) overload of the AutoResizeRows method and pass in a value of false for the Boolean parameter, the overload will calculate the ideal heights and widths for cells in the row, but it will adjust the row heights only. You must then call the AutoResizeColumns method to adjust the column widths to the calculated ideal.

5.3.6 Content-based Sizing Options

The enumerations used by sizing properties and methods have similar values for content-based sizing. With these values, you can limit which cells are used to calculate the preferred sizes. For all sizing enumerations, values with names that refer to displayed cells limit their calculations to cells in displayed rows. Excluding rows is useful to avoid a performance penalty when you are working with a large quantity of rows. You can also restrict calculations to cell values in header or nonheader cells.

5.4 Selection modes

The DataGridView control provides you with a variety of options for configuring how users can select cells, rows, and columns. For example, you can enable single or multiple selection, selection of whole rows or columns when users click cells, or selection of whole rows or columns only when users click their headers, which enables cell selection as well. If you want to provide your own user interface for selection, you can disable ordinary selection and handle all selection programmatically. Additionally, you can enable users to copy the selected values to the Clipboard.

Sometimes you want your application to perform actions based on user selections within a DataGridView control. Depending on the actions, you may want to restrict the kinds of selection that are possible. For example, suppose your application can print a report for the currently selected record. In this case, you may want to configure the DataGridView control so that clicking anywhere within a row always selects the entire row, and so that only one row at a time can be selected.

You can specify the selections allowed by setting the SelectionMode property to one of the following DataGridViewSelectionMode enumeration values.

DataGridViewSelectionMode value	Description
CellSelect	Clicking a cell selects it. Row and column headers cannot be used for selection.
ColumnHeaderSelect	Clicking a cell selects it. Clicking a column header selects the entire column. Column headers cannot be used for sorting.
FullColumnSelect	Clicking a cell or a column header selects the entire column. Column headers cannot be used for sorting.
FullRowSelect	Clicking a cell or a row header selects the entire row.
RowHeaderSelect	Default selection mode. Clicking a cell selects it. Clicking a row header selects the entire row.

Note: Changing the selection mode at run time automatically clears the current selection.

By default, users can select multiple rows, columns, or cells by dragging with the mouse, pressing CTRL or SHIFT while selecting to extend or modify a selection, or clicking the top-left header cell to select all cells in the control. To prevent this behavior, set the MultiSelect property to false.

The FullRowSelect and RowHeaderSelect modes allow users to delete rows by selecting them and pressing the DELETE key. Users can delete rows only when the current cell is not in edit mode, the AllowUserToDeleteRows property is set to true, and the underlying data source supports user-driven row deletion. Note that these settings do not prevent programmatic row deletion.

5.4.1 Programmatic Selection

The current selection mode restricts the behavior of programmatic selection as well as user selection. You can change the current selection programmatically by setting the Selected property of any cells, rows, or columns present in the DataGridView control. You can also select all cells in the control through the SelectAll method, depending on the selection mode. To clear the selection, use the ClearSelection method.

If the MultiSelect property is set to true, you can add DataGridView elements to or remove them from the selection by changing the Selected property of the element. Otherwise, setting the

Selected property to true for one element automatically removes other elements from the selection.

Note that changing the value of the CurrentCell property does not alter the current selection.

You can retrieve a collection of the currently selected cells, rows, or columns through the SelectedCells, SelectedRows, and SelectedColumns properties of the DataGridView control. Accessing these properties is inefficient when every cell in the control is selected. To avoid a performance penalty in this case, use the AreAllCellsSelected method first. Additionally, accessing these collections to determine the number of selected cells, rows, or columns can be inefficient. Instead, you should use the GetCellCount, GetRowCount, or GetColumnCount method, passing in the Selected value.

5.5 Scrolling

The DataGridView obviously provides scrolling support via the horizontal and vertical scrollbars. It also provides vertical scrolling via the mouse wheel. Scrolling horizontally scrolls in pixel values while scrolling vertically scrolls in row index. The DataGridView does not provide support for scrolling rows in pixel increments.

5.5.1 Scroll event

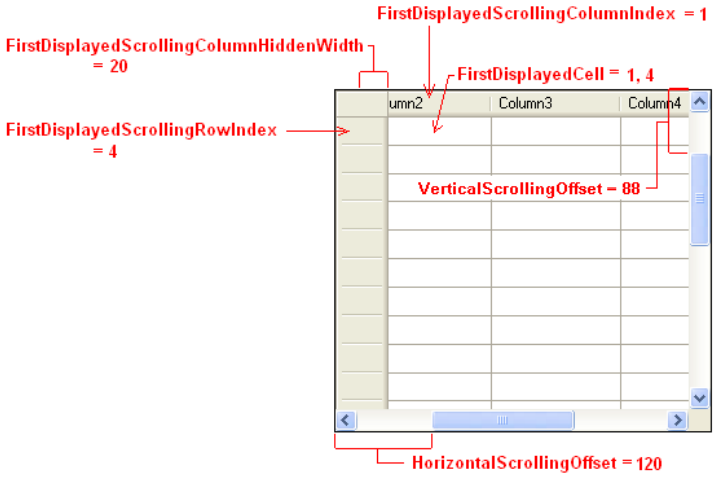
As you scroll the DataGridView raises the Scroll event that allows you to be notified that scrolling is occurring. The Orientation property on the scroll event args lets you know the scroll direction.

5.5.2 Scroll bars

The DataGridView provides access to the scrollbars that it displays via the protected HorizontalScrollBar and VerticalScrollBar properties. Accessing these ScrollBar controls directly allow you to have finer control over scrolling.

5.5.3 Scrolling Properties

There are a set of properties that provide greater level of details on how the DataGridView is scrolled. The diagram highlights these properties and their values at this state. The properties are read/write except for the FirstDisplayedScrollingColumnHiddenWidth and VerticalScrollingOffset properties.



5.6 Sorting

By default, users can sort the data in a DataGridView control by clicking the header of a text box column. You can modify the SortMode property of specific columns to allow users to sort by other column types when it makes sense to do so. You can also sort the data programmatically by any column, or by multiple columns.

DataGridView columns have three sort modes. The sort mode for each column is specified through the SortMode property of the column, which can be set to one of the following DataGridViewColumnSortMode enumeration values.

DataGridViewColumnSortMode value	Description
Automatic	Default for text box columns. Unless column headers are used for selection, clicking the column header automatically sorts the **DataGridView** by this column and displays a glyph indicating the sort order.
NotSortable	Default for non–text box columns. You can sort this column programmatically; however, it is not intended for sorting, so no space is reserved for the sorting glyph.
Programmatic	You can sort this column programmatically, and space is reserved for the sorting glyph.

You might want to change the sort mode for a column that defaults to NotSortable if it contains values that can be meaningfully ordered. For example, if you have a database column containing numbers that represent item states, you can display these numbers as corresponding icons by binding an image column to the database column. You can then change the numerical cell values into image display values in a handler for the CellFormatting event. In this case, setting the SortMode property to Automatic will enable your users to sort the column. Automatic sorting will enable your users to group items that have the same state even if the states corresponding to the

numbers do not have a natural sequence. Check box columns are another example where automatic sorting is useful for grouping items in the same state.

You can sort a DataGridView programmatically by the values in any column or in multiple columns, regardless of the SortMode settings. Programmatic sorting is useful when you want to provide your own user interface (UI) for sorting or when you want to implement custom sorting. Providing your own sorting UI is useful, for example, when you set the DataGridView selection mode to enable column header selection. In this case, although the column headers cannot be used for sorting, you still want the headers to display the appropriate sorting glyph, so you would set the SortMode property to Programmatic.

Columns set to programmatic sort mode do not automatically display a sorting glyph. For these columns, you must display the glyph yourself by setting the DataGridViewColumnHeaderCell.SortGlyphDirection property. This is necessary if you want flexibility in custom sorting. For example, if you sort the DataGridView by multiple columns, you might want to display multiple sorting glyphs or no sorting glyph.

Although you can programmatically sort a DataGridView by any column, some columns, such as button columns, might not contain values that can be meaningfully ordered. For these columns, a SortMode property setting of NotSortable indicates that it will never be used for sorting, so there is no need to reserve space in the header for the sorting glyph.

When a DataGridView is sorted, you can determine both the sort column and the sort order by checking the values of the SortedColumn and SortOrder properties. These values are not meaningful after a custom sorting operation. For more information about custom sorting, see the Custom Sorting section later in this topic.

When a DataGridView control containing both bound and unbound columns is sorted, the values in the unbound columns cannot be maintained automatically. To maintain these values, you must implement virtual mode by setting the VirtualMode property to true and handling the CellValueNeeded and CellValuePushed events.

5.6.1 Programmatic Sorting

You can sort a DataGridView programmatically by calling its Sort method.

The Sort(DataGridViewColumn,ListSortDirection) overload of the Sort method takes a DataGridViewColumn and a ListSortDirection enumeration value as parameters. This overload is useful when sorting by columns with values that can be meaningfully ordered, but which you do not want to configure for automatic sorting. When you call this overload and pass in a column with a SortMode property value of DataGridViewColumnSortMode.Automatic, the SortedColumn and SortOrder properties are set automatically and the appropriate sorting glyph appears in the column header.

> **Note:** When the DataGridView control is bound to an external data source by setting the DataSource property, the Sort(DataGridViewColumn,ListSortDirection) method overload does not work for unbound columns. Additionally, when the VirtualMode property is true, you can call this overload only for bound columns. To determine whether a column is data-bound, check the IsDataBound property value. Sorting unbound columns in bound mode is not supported.

5.6.2 Custom Sorting

You can customize DataGridView by using the Sort(IComparer) overload of the Sort method or by handling the SortCompare event.

The Sort(IComparer) method overload takes an instance of a class that implements the IComparer interface as a parameter. This overload is useful when you want to provide custom sorting; for example, when the values in a column do not have a natural sort order or when the natural sort order is inappropriate. In this case, you cannot use automatic sorting, but you might still want your users to sort by clicking the column headers. You can call this overload in a handler for the ColumnHeaderMouseClick event if you do not use column headers for selection.

> **Note:** The Sort(IComparer) method overload works only when the DataGridView control is not bound to an external data source and the VirtualMode property value is false. To customize sorting for columns bound to an external data source, you must use the sorting operations provided by the data source. In virtual mode, you must provide your own sorting operations for unbound columns.

To use the Sort(IComparer) method overload, you must create your own class that implements the IComparer interface. This interface requires your class to implement the IComparer.Compare(Object) method, to which the DataGridView passes DataGridViewRow objects as input when the Sort(IComparer) method overload is called. With this, you can calculate the correct row ordering based on the values in any column.

The Sort(IComparer) method overload does not set the SortedColumn and SortOrder properties, so you must always set the DataGridViewColumnHeaderCell.SortGlyphDirection property to display the sorting glyph.

As an alternative to the Sort(IComparer) method overload, you can provide custom sorting by implementing a handler for the SortCompare event. This event occurs when users click the headers of columns configured for automatic sorting or when you call the Sort(DataGridViewColumn,ListSortDirection) overload of the Sort method. The event occurs for each pair of rows in the control, enabling you to calculate their correct order.

> **Note:** The SortCompare event does not occur when the DataSource property is set or when the VirtualMode property value is true.

Common questions and scenarios

1) How do I prevent sorting?
2) How do I sort on multiple columns?

5.7 Border styles

With the DataGridView control, you can customize the appearance of the control's border and gridlines to improve the user experience. You can modify the gridline color and the control border style in addition to the border styles for the cells within the control. The gridline color is controlled via the GridColor property. You can also apply different cell border styles for ordinary cells, row header cells, and column header cells. For advanced border styles the DataGridView provides the advanced border style properties as well.

> **Note:** The gridline color is used only with the Single, SingleHorizontal, and SingleVertical values of the DataGridViewCellBorderStyle enumeration and the Single value of the DataGridViewHeaderBorderStyle enumeration. The other values of these enumerations use colors specified by the operating system. Additionally, when visual styles are enabled on Windows XP and above, the GridColor property value is not used.

5.7.1 Standard Border Styles

Standard border styles are controlled via the CellBorderStyle, RowHeadersBorderStyle, and ColumnHeadersBorderStyle properties.

The following table identifies the standard border styles available via the :

BorderStyle value	Description
Fixed3D	A three-dimensional border.
FixedSingle	A single-line border.
None	No border.

5.7.2 Advanced Border Styles

The DataGridView control allows you to fully customize its appearance, including the borders of the cells and headers. The DataGridView has CellBorderStyle, ColumnHeadersBorderStyle, and RowHeadersBorderStyle properties that allow you to set the appearance of the cell border. However, if you need to further customize the borders, the DataGridViewAdvancedBorderStyle class allows you to set the style of the border on the individual sides of the cells. The Left, Right, Top, and Bottom properties of DataGridViewAdvancedBorderStyle represent the left, right, top, and bottom border of a cell, respectively. You can set these properties on the AdvancedCellBorderStyle, AdvancedColumnHeadersBorderStyle, AdvancedRowHeadersBorderStyle properties of the DataGridView to produce various appearances for the borders between the cells.

The following table identifies the advanced border styles available that can be set for the left, right, top and bottom parts. Note that some combinations are not valid.

BorderStyle value	Description
Inset	A three-dimensional border.
InsetDouble	A single-line border.
None	No border.
NotSet	The border is not set
Outset	A single-line raised border
OutsetDouble	A double-line raised border
OutsetPartial	A single-line border containing a raised portion
Single	A single-line border

5.8 Enter-Edit modes

By default, users can edit the contents of the current DataGridView text box cell by typing in it or pressing F2. This puts the cell in edit mode if all of the following conditions are met:

- The underlying data source supports editing.
- The DataGridView control is enabled.
- The EditMode property value is not EditProgrammatically.
- The ReadOnly properties of the cell, row, column, and control are all set to false.

In edit mode, the user can change the cell value and press ENTER to commit the change or ESC to revert the cell to its original value.

You can configure a DataGridView control so that a cell enters edit mode as soon as it becomes the current cell. The behavior of the ENTER and ESC keys is unchanged in this case, but the cell remains in edit mode after the value is committed or reverted. You can also configure the control so that cells enter edit mode only when users type in the cell or only when users press F2. Finally, you can prevent cells from entering edit mode except when you call the BeginEdit method.

The following table describes the different edit modes available:

EditMode value	Description
EditOnEnter	Editing begins when the cell receives focus. This mode is useful when pressing the TAB key to enter values across a row, or when pressing the ENTER key to enter values down a column.
EditOnF2	Editing begins when F2 is pressed while the cell has focus. This mode places the selection point at the end of the cell contents.
EditOnKeystroke	Editing begins when any alphanumeric key is pressed while the cell has focus.
EditOnKeystrokeOrF2	Editing begins when any alphanumeric key or F2 is pressed while the cell has focus.
EditProgrammatically	Editing begins only when the BeginEdit method is called.

5.9 Clipboard copy modes

When you enable cell copying, you make the data in your DataGridView control easily accessible to other applications through the Clipboard. The DataGridView control copies the text representation of each selected cell to the Clipboard. This value is the cell value converted to a string or, for image cells, the value of the Description property. The content is then added to the Clipboard as tab-delimited text values for pasting into applications like Notepad and Excel, and as an HTML-formatted table for pasting into applications like Word.

You can configure cell copying to copy cell values only, to include row and column header text in the Clipboard data, or to include header text only when users select entire rows or columns.

The following table identifies the different clipboard copy modes:

Clipboard Copy modes	Description
Disable	Copying to the Clipboard is disabled.
EnableAlwaysIncludeHeaderText	The text values of selected cells can be copied to the Clipboard. Header text is included for rows and columns that contain selected cells.
EnableWithAutoHeaderText	The text values of selected cells can be copied to the Clipboard. Row or column header text is included for rows or columns that contain selected cells only when the SelectionMode property is set to RowHeaderSelect or ColumnHeaderSelect and at least one header is selected.
EnableWithoutHeaderText	The text values of selected cells can be copied to the Clipboard. Header text is not included.

Depending on the selection mode, users can select multiple disconnected groups of cells. When a user copies cells to the Clipboard, rows and columns with no selected cells are not copied. All

other rows or columns become rows and columns in the table of data copied to the Clipboard. Unselected cells in these rows or columns are copied as blank placeholders to the Clipboard.

When users copy content, the DataGridView control adds a DataObject to the Clipboard. This data object is retrieved from the GetClipboardContent() method. You can call this method when you want to programmatically add the data object to the Clipboard. The GetClipboardContent() method retrieves values for individual cells by calling the DataGridViewCell.GetClipboardContent() method. You can override either or both of these methods in derived classes to customize the layout of copied cells or to support additional data formats.

5.10 Frozen columns/rows

When users view data sometimes they need to refer to a single column or set of columns frequently. For example, when displaying a table of customer information that contains many columns, it is useful to display the customer name at all times while enabling other columns to scroll outside the visible region.

To achieve this behavior, you can freeze columns in the control. This is done via setting the Frozen property on the column or row. When you freeze a column, all the columns to its left (or to its right in right-to-left language scripts) are frozen as well. Frozen columns remain in place while all other columns can scroll. Rows act in similar fashion: all rows before the frozen row are frozen as well and remain in place while the non frozen rows can scroll.

5.11 Implementing Custom cells and editing controls/cells

You can implement the IDataGridViewEditingCell interface in your derived cell class to create a cell type that has editing functionality but does not host a control in editing mode. To create a control that you can host in a cell in editing mode, you can implement the IDataGridViewEditingControl interface in a class derived from Control.

5.11.1 IDataGridViewEditingControl

Cells that support advanced editing functionality typically use a hosted control that is derived from a Windows Forms control. This interface is implemented by editing controls, such as DataGridViewComboBoxEditingControl and DataGridViewTextBoxEditingControl, that are hosted by the corresponding DataGridView cells, such as DataGridViewComboBoxCell and DataGridViewTextBoxCell, when they are in edit mode.

Cell types that can that host editing controls set their EditType property to a Type representing the editing control type.

5.11.2 IDataGridViewEditingCell

This interface is implemented by classes to provide a user interface (UI) for specifying values without hosting an editing control. The UI in this case is displayed regardless of whether the cell is in edit mode. The DataGridViewCheckBoxCell is an example of a cell that implements the IDataGridViewEditingCell interface.

Other cell types, such as DataGridViewButtonCell, provide a UI but do not store user-specified values. In this case, the cell type does not implement IDataGridViewEditingCell or host an editing control.

5.12 Virtual mode

With virtual mode, you can manage the interaction between the DataGridView control and a custom data cache. To implement virtual mode, set the VirtualMode property to true and handle

one or more of the events described in this topic. You will typically handle at least the CellValueNeeded event, which enables the control look up values in the data cache.

5.12.1 Bound Mode and Virtual Mode

Virtual mode is necessary only when you need to supplement or replace bound mode. In bound mode, you set the DataSource property and the control automatically loads the data from the specified source and submits user changes back to it. You can control which of the bound columns are displayed, and the data source itself typically handles operations such as sorting.

5.12.2 Supplementing Bound Mode

You can supplement bound mode by displaying unbound columns along with the bound columns. This is sometimes called "mixed mode" and is useful for displaying things like calculated values or user-interface (UI) controls.
Because unbound columns are outside the data source, they are ignored by the data source's sorting operations. Therefore, when you enable sorting in mixed mode, you must manage the unbound data in a local cache and implement virtual mode to let the DataGridView control interact with it.

Common questions and scenarios

1) How do I show unbound data along with bound data?
2) How do I show data that comes from two tables?

5.12.3 Replacing Bound Mode

If bound mode does not meet your performance needs, you can manage all your data in a custom cache through virtual-mode event handlers. For example, you can use virtual mode to implement a just-in-time data loading mechanism that retrieves only as much data from a networked database as is necessary for optimal performance. This scenario is particularly useful when working with large amounts of data over a slow network connection or with client machines that have a limited amount of RAM or storage space.

5.12.4 Virtual-Mode Events

If your data is read-only, the CellValueNeeded event may be the only event you will need to handle. Additional virtual-mode events let you enable specific functionality like user edits, row addition and deletion, and row-level transactions.

Some standard DataGridView events (such as events that occur when users add or delete rows, or when cell values are edited, parsed, validated, or formatted) are useful in virtual mode, as well. You can also handle events that let you maintain values not typically stored in a bound data source, such as cell ToolTip text, cell and row error text, cell and row shortcut menu data, and row height data.

The following events occur only when the VirtualMode property is set to true.

Event	Description
CellValueNeeded	Used by the control to retrieve a cell value from the data cache for display. This event occurs only for cells in unbound columns.
CellValuePushed	Used by the control to commit user input for a cell to the data cache. This event occurs only for cells in unbound columns. Call the UpdateCellValue method when changing a cached value outside of a **CellValuePushed** event handler to ensure that the current value is displayed

	in the control and to apply any automatic sizing modes currently in effect.
NewRowNeeded	Used by the control to indicate the need for a new row in the data cache.
RowDirtyStateNeeded	Used by the control to determine whether a row has any uncommitted changes.
CancelRowEdit	Used by the control to indicate that a row should revert to its cached values.

The following events are useful in virtual mode, but can be used regardless of the VirtualMode property setting.

Events	Description
UserDeletingRow **UserDeletedRow** **RowsRemoved** **RowsAdded**	Used by the control to indicate when rows are deleted or added, letting you update the data cache accordingly.
CellFormatting **CellParsing** **CellValidating** **CellValidated** **RowValidating** **RowValidated**	Used by the control to format cell values for display and to parse and validate user input.
CellToolTipTextNeeded	Used by the control to retrieve cell ToolTip text when the **DataSource** property is set or the **VirtualMode** property is **true**. Cell ToolTips are displayed only when the ShowCellToolTips property value is **true**.
CellErrorTextNeeded **RowErrorTextNeeded**	Used by the control to retrieve cell or row error text when the **DataSource** property is set or the **VirtualMode** property is **true**. Call the UpdateCellErrorText method or the UpdateRowErrorText method when you change the cell or row error text to ensure that the current value is displayed in the control. Cell and row error glyphs are displayed when the ShowCellErrors and ShowRowErrors property values are **true**.
CellContextMenuStripNeeded **RowContextMenuStripNeeded**	Used by the control to retrieve a cell or row ContextMenuStrip when the control **DataSource** property is set or the **VirtualMode** property is **true**.
RowHeightInfoNeeded **RowHeightInfoPushed**	Used by the control to retrieve or store row height information in the data cache. Call the UpdateRowHeightInfo method when changing the cached row height information outside of a **RowHeightInfoPushed** event handler to ensure that the current value is used in the display of the control.

5.12.5 Best Practices in Virtual Mode

If you are implementing virtual mode in order to work efficiently with large amounts of data, you will also want to ensure that you are working efficiently with the DataGridView control itself. See below for more information on best practices

5.13 Capacity

In general, the DataGridView does not have any hard-coded capacity limits. The grid was designed so that more and more content can be added as machines become faster and have more memory. That said, the grid was not designed to deal with large number of columns. If you add more than 300 columns you will start to notice a degradation in performance as our performance tuning of the grid was not designed for this. If you need a grid with large amounts of columns then the DataGridView might not meet your needs. Regarding the number of rows supported, the DataGridView is bound by memory constraints. When using Virtual mode you can easily support over 2 million rows. Check out the best practices section below for information on things you can do (and not do) to improve memory usage and performance.

6 Best Practices

The DataGridView control is designed to provide maximum scalability. If you need to display large amounts of data, you should follow the guidelines described in this topic to avoid consuming large amounts of memory or degrading the responsiveness of the user interface (UI).

6.1 Using Cell Styles Efficiently

Each cell, row, and column can have its own style information. Style information is stored in DataGridViewCellStyle objects. Creating cell style objects for many individual DataGridView elements can be inefficient, especially when working with large amounts of data. To avoid a performance impact, use the following guidelines:

- Avoid setting cell style properties for individual DataGridViewCell or DataGridViewRow objects. This includes the row object specified by the RowTemplate property. Each new row that is cloned from the row template will receive its own copy of the template's cell style object. For maximum scalability, set cell style properties at the DataGridView level. For example, set the DefaultCellStyle property rather than the DataGridViewCell.Style property.

- If some cells require formatting other than default formatting, use the same DataGridViewCellStyle instance across groups of cells, rows, or columns. Avoid directly setting properties of type DataGridViewCellStyle on individual cells, rows, and columns. For an example of cell style sharing, see How to: Set Default Cell Styles for the Windows Forms DataGridView Control. You can also avoid a performance penalty when setting cell styles individually by handling the CellFormatting event handler. For an example, see How to: Customize Data Formatting in the Windows Forms DataGridView Control.

- When determining a cell's style, use the DataGridViewCell.InheritedStyle property rather than the DataGridViewCell.Style property. Accessing the Style property creates a new instance of the DataGridViewCellStyle class if the property has not already been used. Additionally, this object might not contain the complete style information for the cell if some styles are inherited from the row, column, or control. For more information about cell style inheritance, see Cell Styles in the Windows Forms DataGridView Control.

6.2 Using Shortcut Menus Efficiently

Each cell, row, and column can have its own shortcut menu. Shortcut menus in the DataGridView control are represented by ContextMenuStrip controls. Just as with cell style objects, creating shortcut menus for many individual DataGridView elements will negatively impact performance. To avoid this penalty, use the following guidelines:

- Avoid creating shortcut menus for individual cells and rows. This includes the row template, which is cloned along with its shortcut menu when new rows are added to the control. For maximum scalability, use only the control's ContextMenuStrip property to specify a single shortcut menu for the entire control.

- If you require multiple shortcut menus for multiple rows or cells, handle the CellContextMenuStripNeeded or RowContextMenuStripNeeded events. These events let you manage the shortcut menu objects yourself, allowing you to tune performance.

6.3 Using Automatic Resizing Efficiently

Rows, columns, and headers can be automatically resized as cell content changes so that the entire contents of cells are displayed without clipping. Changing sizing modes can also resize rows, columns, and headers. To determine the correct size, the DataGridView control must examine the value of each cell that it must accommodate. When working with large data sets, this analysis can negatively impact the performance of the control when automatic resizing occurs. To avoid performance penalties, use the following guidelines:

- Avoid using automatic sizing on a DataGridView control with a large set of rows. If you do use automatic sizing, only resize based on the displayed rows. Use only the displayed rows in virtual mode as well.
- For rows and columns, use the DisplayedCells or DisplayedCellsExceptHeaders field of the DataGridViewAutoSizeRowsMode, DataGridViewAutoSizeColumnsMode, and DataGridViewAutoSizeColumnMode enumerations.
- For row headers, use the AutoSizeToDisplayedHeaders or AutoSizeToFirstHeader field of the DataGridViewRowHeadersWidthSizeMode enumeration.
- For maximum scalability, turn off automatic sizing and use programmatic resizing.

6.4 Using the Selected Cells, Rows, and Columns Collections Efficiently

The SelectedCells collection does not perform efficiently with large selections. The SelectedRows and SelectedColumns collections can also be inefficient, although to a lesser degree because there are many fewer rows than cells in a typical DataGridView control, and many fewer columns than rows. To avoid performance penalties when working with these collections, use the following guidelines:

- To determine whether all the cells in the DataGridView have been selected before you access the contents of the SelectedCells collection, check the return value of the AreAllCellsSelected method. Note, however, that this method can cause rows to become unshared. For more information, see the next section.
- Avoid using the Count property of the DataGridViewSelectedCellCollection to determine the number of selected cells. Instead, use the GetCellCount() method and pass in the DataGridViewElementStates.Selected value. Similarly, use the DataGridViewRowCollection.GetRowCount() and DataGridViewColumnCollection.GetColumnCount() methods to determine the number of selected elements, rather than accessing the selected row and column collections.
- Avoid cell-based selection modes. Instead, set the SelectionMode property to FullRowSelect or FullColumnSelect.

6.5 Using Shared Rows

Efficient memory use is achieved in the DataGridView control through shared rows. Rows will share as much information about their appearance and behavior as possible by sharing instances of the DataGridViewRow class.

While sharing row instances saves memory, rows can easily become unshared. For example, whenever a user interacts directly with a cell, its row becomes unshared. Because this cannot be avoided, the guidelines in this topic are useful only when working with very large amounts of data and only when users will interact with a relatively small part of the data each time your program is run.

A row cannot be shared in an unbound DataGridView control if any of its cells contain values. When the DataGridView control is bound to an external data source or when you implement virtual mode and provide your own data source, the cell values are stored outside the control rather than in cell objects, allowing the rows to be shared.

A row object can only be shared if the state of all its cells can be determined from the state of the row and the states of the columns containing the cells. If you change the state of a cell so that it can no longer be deduced from the state of its row and column, the row cannot be shared.

For example, a row cannot be shared in any of the following situations:

- The row contains a single selected cell that is not in a selected column.
- The row contains a cell with its ToolTipText or ContextMenuStrip properties set.
- The row contains a DataGridViewComboBoxCell with its Items property set.

In bound mode or virtual mode, you can provide ToolTips and shortcut menus for individual cells by handling the CellToolTipTextNeeded and CellContextMenuStripNeeded events.

The DataGridView control will automatically attempt to use shared rows whenever rows are added to the DataGridViewRowCollection. Use the following guidelines to ensure that rows are shared:

- Avoid calling the Add(Object[]) overload of the Add method and the Insert(Object[]) overload of the Insert method of the Rows collection. These overloads automatically create unshared rows.
- Be sure that the row specified in the RowTemplate property can be shared in the following cases:
  - When calling the Add() or Add(Int) overloads of the Add method or the Insert(Int, Int) overload of the Insert method of the Rows collection.
  - When increasing the value of the RowCount property.
  - When setting the DataSource property.
- Be sure that the row indicated by the indexSource parameter can be shared when calling the AddCopy, AddCopies, InsertCopy, and InsertCopies methods of the Rows collection.
- Be sure that the specified row or rows can be shared when calling the Add(DataGridViewRow) overload of the Add method, the AddRange method, the Insert(Int32,DataGridViewRow) overload of the Insert method, and the InsertRange method of the Rows collection.

To determine whether a row is shared, use the DataGridViewRowCollection.SharedRow(Int) method to retrieve the row object, and then check the object's Index property. Shared rows always have an Index property value of –1.

6.6 Preventing Rows from Becoming Unshared

Shared rows can become unshared as a result of code or user action. To avoid a performance impact, you should avoid causing rows to become unshared. During application development, you can handle the RowUnshared event to determine when rows become unshared. This is useful when debugging row-sharing problems.

To prevent rows from becoming unshared, use the following guidelines:

- Avoid indexing the Rows collection or iterating through it with a foreach loop. You will not typically need to access rows directly. DataGridView methods that operate on rows take row index arguments rather than row instances. Additionally, handlers for row-related events receive event argument objects with row properties that you can use to manipulate rows without causing them to become unshared.
- If you need to access a row object, use the DataGridViewRowCollection.SharedRow(Int) method and pass in the row's actual index. Note, however, that modifying a shared row object retrieved through this method will modify all the rows that share this object. The row for new records is not shared with other rows, however, so it will not be affected when you modify any other row. Note also that different rows represented by a shared row may have different shortcut menus. To retrieve the correct shortcut menu from a shared row instance, use the GetContextMenuStrip method and pass in the row's actual index. If you access the shared row's ContextMenuStrip property instead, it will use the shared row index of -1 and will not retrieve the correct shortcut menu.

- Avoid indexing the DataGridViewRow.Cells collection. Accessing a cell directly will cause its parent row to become unshared, instantiating a new DataGridViewRow. Handlers for cell-related events receive event argument objects with cell properties that you can use to manipulate cells without causing rows to become unshared. You can also use the CurrentCellAddress property to retrieve the row and column indexes of the current cell without accessing the cell directly.
- Avoid cell-based selection modes. These modes cause rows to become unshared. Instead, set the SelectionMode property to DataGridViewSelectionMode.FullRowSelect or DataGridViewSelectionMode.FullColumnSelect.
- Do not handle the DataGridViewRowCollection.CollectionChanged or RowStateChanged events. These events cause rows to become unshared. Also, do not call the DataGridViewRowCollection.OnCollectionChanged(CollectionChangeEventArgs) or OnRowStateChanged(Int,DataGridViewRowStateChangedEventArgs) methods, which raise these events.
- Do not access the SelectedCells collection when the SelectionMode property value is FullColumnSelect, ColumnHeaderSelect, FullRowSelect, or RowHeaderSelect. This causes all selected rows to become unshared.
- Do not call the AreAllCellsSelected(boolean) method. This method can cause rows to become unshared.
- Do not call the SelectAll method when the SelectionMode property value is CellSelect. This causes all rows to become unshared.
- Do not set the ReadOnly or Selected property of a cell to false when the corresponding property in its column is set to true. This causes all rows to become unshared.
- Do not access the DataGridViewRowCollection.List property. This causes all rows to become unshared.
- Do not call the Sort(IComparer) overload of the Sort method. Sorting with a custom comparer causes all rows to become unshared.

Appendix A – Common Questions and Answers

The following appendix contains code samples and snippets that are a part of the common questions

1. How do I prevent a particular cell from being editable?

The ReadOnly property indicates whether the data displayed by the cell can be edited. You can set ReadOnly for individual cells, or you can make an entire row or column of cells read-only by setting the DataGridViewRow.ReadOnly or DataGridViewColumn.ReadOnly properties. By default, if a cell's parent row or column is set to read-only, the child cells will adopt the same value.

While you can navigate to a read-only cell, and you can set a read-only cell to be the current cell, the content cannot be modified by the user. Note that the ReadOnly property does not keep the cell from being modified programmatically. Also note that ReadOnly does not affect whether the user can delete rows

2. How do I disable a cell?

While a cell can be read-only to prevent it from being editable, the DataGridView does not have built-in support for disabling a cell. Normally the concept of “disabled” means that the user cannot navigate to it and usually has a visual cue that it is disabled. There isn’t any easy way to create the navigational side of disabled, but the visual cue is something that can be done. While none of the built-in cell have a disabled property, the following example extends the DataGridViewButtonCell and implements a visual “disabled” state along with a corresponding disabled property.

```
public class DataGridViewDisableButtonColumn : DataGridViewButtonColumn
{
    public DataGridViewDisableButtonColumn()
    {
        this.CellTemplate = new DataGridViewDisableButtonCell();
    }
}

public class DataGridViewDisableButtonCell : DataGridViewButtonCell
{
    private bool enabledValue;
    public bool Enabled
    {
        get {
            return enabledValue;
        }
        set {
            enabledValue = value;
        }
    }

    // Override the Clone method so that the Enabled property is copied.
    public override object Clone()
    {
        DataGridViewDisableButtonCell cell =
            (DataGridViewDisableButtonCell)base.Clone();
        cell.Enabled = this.Enabled;
        return cell;
    }

    // By default, enable the button cell.
```

```
public DataGridViewDisableButtonCell()
{
    this.enabledValue = true;
}

protected override void Paint(Graphics graphics,
    Rectangle clipBounds, Rectangle cellBounds, int rowIndex,
    DataGridViewElementStates elementState, object value,
    object formattedValue, string errorText,
    DataGridViewCellStyle cellStyle,
    DataGridViewAdvancedBorderStyle advancedBorderStyle,
    DataGridViewPaintParts paintParts)
{
    // The button cell is disabled, so paint the border,
    // background, and disabled button for the cell.
    if (!this.enabledValue)
    {
        // Draw the cell background, if specified.
        if ((paintParts & DataGridViewPaintParts.Background) ==
            DataGridViewPaintParts.Background)
        {
            SolidBrush cellBackground =
                new SolidBrush(cellStyle.BackColor);
            graphics.FillRectangle(cellBackground, cellBounds);
            cellBackground.Dispose();
        }

        // Draw the cell borders, if specified.
        if ((paintParts & DataGridViewPaintParts.Border) ==
            DataGridViewPaintParts.Border)
        {
            PaintBorder(graphics, clipBounds, cellBounds, cellStyle,
                advancedBorderStyle);
        }

        // Calculate the area in which to draw the button.
        Rectangle buttonArea = cellBounds;
        Rectangle buttonAdjustment =
            this.BorderWidths(advancedBorderStyle);
        buttonArea.X += buttonAdjustment.X;
        buttonArea.Y += buttonAdjustment.Y;
        buttonArea.Height -= buttonAdjustment.Height;
        buttonArea.Width -= buttonAdjustment.Width;

        // Draw the disabled button.
        ButtonRenderer.DrawButton(graphics, buttonArea,
            PushButtonState.Disabled);

        // Draw the disabled button text.
        if (this.FormattedValue is String)
        {
            TextRenderer.DrawText(graphics,
                (string)this.FormattedValue,
                this.DataGridView.Font,
                buttonArea, SystemColors.GrayText);
        }
    }
    else
    {
        // The button cell is enabled, so let the base class
        // handle the painting.
        base.Paint(graphics, clipBounds, cellBounds, rowIndex,
            elementState, value, formattedValue, errorText,
```



```
            cellStyle, advancedBorderStyle, paintParts);
    }
}
```

3. How do I restrict user from setting focus to a specific cell?

By default the DataGridView's navigation model does not have any ability to restrict focus to a specific cell. You can implement your own navigation logic by overriding the appropriate keyboard, navigation and mouse methods such as DataGridView.OnKeyDown, DataGridView.ProcessDataGridViewKey, DataGridView.SetCurrentCellAddressCore, DataGridView.SetSelectedCellCore, DataGridView.OnMouseDown.

4. How do I show controls in all cells regardless of edit?

The DataGridView control only supports displaying an actual control when a cell is in edit mode. The DataGridView control is not designed to display multiple controls or repeat a set of controls per row. The DataGridView control draws a representation of the control when the cell is not being edited. This representation can be a detailed as you want. For example, the DataGridViewButtonCell draws a button regardless of the cell being in edit or not.

5. Why does the cell text show up with "square" characters where they should be new lines?

By default, text in a DataGridViewTextBoxCell does not wrap. This can be controlled via the WrapMode property on the cell style (e.g. DataGridView.DefaultCellStyle.WrapMode). Because text doesn't wrap, new line characters in the text do not apply and so they are displayed as a "non-printable" character. This is similar to setting a TextBox's Text property to the same text when the TextBox's MultiLine property is false.

6. How do I show icon and text in the same cell?

The DataGridView control does not have any built-in support for showing an icon and text in the same cell. Through the different painting customization events, such as the CellPaint event, you can easily display an icon next to the text in the cell.

The following example extends the DataGridViewTextColumn and cell to paint an image next to the text. The sample uses the DataGridViewCellStyle.Padding property to adjust the text location and overrides the Paint method to paint an icon. This sample can be simplified by handling the CellPainting event and performing similar code.

```
public class TextAndImageColumn:DataGridViewTextBoxColumn
{
    private Image imageValue;
    private Size imageSize;

    public TextAndImageColumn()
    {
        this.CellTemplate = new TextAndImageCell();
    }

    public override object Clone()
    {
        TextAndImageColumn c = base.Clone() as TextAndImageColumn;
        c.imageValue = this.imageValue;
        c.imageSize = this.imageSize;
        return c;
    }
```

```
    public Image Image
    {
        get { return this.imageValue; }
        set
        {
            if (this.Image != value) {
                this.imageValue = value;
                this.imageSize = value.Size;

                if (this.InheritedStyle != null) {
                    Padding inheritedPadding = this.InheritedStyle.Padding;
                    this.DefaultCellStyle.Padding = new Padding(imageSize.Width,
                              inheritedPadding.Top, inheritedPadding.Right,
                              inheritedPadding.Bottom);
                }
            }
        }
    }
    private TextAndImageCell TextAndImageCellTemplate
    {
        get { return this.CellTemplate as TextAndImageCell; }
    }
    internal Size ImageSize
    {
        get { return imageSize; }
    }
}

public class TextAndImageCell : DataGridViewTextBoxCell
{
    private Image imageValue;
    private Size imageSize;

    public override object Clone()
    {
        TextAndImageCell c = base.Clone() as TextAndImageCell;
        c.imageValue= this.imageValue;
        c.imageSize = this.imageSize;
        return c;
    }

    public Image Image
    {
        get {
            if (this.OwningColumn == null ||
                this.OwningTextAndImageColumn == null) {

                return imageValue;
            }
            else if (this.imageValue != null) {
                return this.imageValue;
            }
            else {
                return this.OwningTextAndImageColumn.Image;
            }
        }
        set {
            if (this.imageValue != value) {
                this.imageValue = value;
                this.imageSize = value.Size;

                Padding inheritedPadding = this.InheritedStyle.Padding;
```



```
                this.Style.Padding = new Padding(imageSize.Width,
                            inheritedPadding.Top, inheritedPadding.Right,
                            inheritedPadding.Bottom);
            }
        }
    }
    protected override void Paint(Graphics graphics, Rectangle clipBounds,
          Rectangle cellBounds, int rowIndex, DataGridViewElementStates cellState,
          object value, object formattedValue, string errorText,
          DataGridViewCellStyle cellStyle,
          DataGridViewAdvancedBorderStyle advancedBorderStyle,
          DataGridViewPaintParts paintParts)
    {
        // Paint the base content
        base.Paint(graphics, clipBounds, cellBounds, rowIndex, cellState,
                   value, formattedValue, errorText, cellStyle,
                   advancedBorderStyle, paintParts);

        if (this.Image != null) {
            // Draw the image clipped to the cell.
            System.Drawing.Drawing2D.GraphicsContainer container =
                       graphics.BeginContainer();

            graphics.SetClip(cellBounds);
            graphics.DrawImageUnscaled(this.Image, cellBounds.Location);

            graphics.EndContainer(container);
        }
    }

    private TextAndImageColumn OwningTextAndImageColumn
    {
        get { return this.OwningColumn as TextAndImageColumn; }
    }
}
```

7. How do I hide a column?

Sometimes you will want to display only some of the columns that are available in a DataGridView. For example, you might want to show an employee salary column to users with management credentials while hiding it from other users.

To hide a column programmatically
In the DataGridView control, the column's Visible property determines whether the column is displayed.

To hide a column using the designer

1) Choose Edit Columns from the control's smart tag.
2) Select a column from the Selected Columns list.
3) In the Column Properties grid, set the Visible property to false.

8. How do I prevent the user from sorting on a column?

In the DataGridView control, text box columns use automatic sorting by default, while other column types are not sorted automatically. Sometimes you will want to override these defaults.

In the DataGridView control, the SortMode property value of a column determines its sorting behavior.

9. How do I sort on multiple columns?

By default the DataGridView control does not provide sorting on multiple columns. Depending upon if the DataGridView is databound or not, you can provide additional support for sorting on multiple columns.

9.1 Databound DataGridView

When the DataGridView is databound the datasource can be sorted on multiple columns and the DataGridView will respect that sorting, but the only the first sorted column will display the sort glyph. In addition, the SortedColumn property will only return the first sorted column.

Some datasources have built in support for sorting on multiple columns. If your datasource implements IBindingListView and provides support for the Sort property, then using it will provide support for multi-column sorting. To indicate in the grid that multiple columns are sorted on, manually set a column's SortGlyphDirection to properly indicate that the column is sorted.

The following example uses a DataTable and sets the default view's Sort property to sort on the second and third columns. The example also demonstrates setting the column's SortGlyphDirection. The example assumes that you have a DataGridView and a BindingSource component on your form:

```
DataTable dt = new DataTable();
dt.Columns.Add("C1", typeof(int));
dt.Columns.Add("C2", typeof(string));
dt.Columns.Add("C3", typeof(string));

dt.Rows.Add(1, "1", "Test1");
dt.Rows.Add(2, "2", "Test2");
dt.Rows.Add(2, "2", "Test1");
dt.Rows.Add(3, "3", "Test3");
dt.Rows.Add(4, "4", "Test4");
dt.Rows.Add(4, "4", "Test3");

DataView view = dt.DefaultView;
view.Sort = "C2 ASC, C3 ASC";
bindingSource.DataSource = view;

DataGridViewTextBoxColumn col0 = new DataGridViewTextBoxColumn();
col0.DataPropertyName = "C1";
dataGridView1.Columns.Add(col0);
col0.SortMode = DataGridViewColumnSortMode.Programmatic;
col0.HeaderCell.SortGlyphDirection = SortOrder.None;

DataGridViewTextBoxColumn col1 = new DataGridViewTextBoxColumn();
col1.DataPropertyName = "C2";
dataGridView1.Columns.Add(col1);
col1.SortMode = DataGridViewColumnSortMode.Programmatic;
col1.HeaderCell.SortGlyphDirection = SortOrder.Ascending;

DataGridViewTextBoxColumn col2 = new DataGridViewTextBoxColumn();
col2.DataPropertyName = "C3";
dataGridView1.Columns.Add(col2);
col2.SortMode = DataGridViewColumnSortMode.Programmatic;
col2.HeaderCell.SortGlyphDirection = SortOrder.Ascending;
```

9.2 Unbound DataGridView

To provide support for sorting on multiple columns you can handle the SortCompare event or call the Sort(IComparer) overload of the Sort method for greater sorting flexibility.

9.2.1 Custom Sorting Using the SortCompare Event

The following code example demonstrates custom sorting using a SortCompare event handler. The selected DataGridViewColumn is sorted and, if there are duplicate values in the column, the ID column is used to determine the final order.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Windows.Forms;

class Form1 : Form
{
    private DataGridView dataGridView1 = new DataGridView();

    // Establish the main entry point for the application.
    [STAThreadAttribute()]
    static void Main()
    {
        Application.EnableVisualStyles();
        Application.Run(new Form1());
    }

    public Form1()
    {
        // Initialize the form.
        // This code can be replaced with designer generated code.
        dataGridView1.AllowUserToAddRows = false;
        dataGridView1.Dock = DockStyle.Fill;
        dataGridView1.SortCompare += new DataGridViewSortCompareEventHandler(
            this.dataGridView1_SortCompare);
        Controls.Add(this.dataGridView1);
        this.Text = "DataGridView.SortCompare demo";

        PopulateDataGridView();
    }

    // Replace this with your own population code.
    public void PopulateDataGridView()
    {
        // Add columns to the DataGridView.
        dataGridView1.ColumnCount = 3;

        // Set the properties of the DataGridView columns.
        dataGridView1.Columns[0].Name = "ID";
        dataGridView1.Columns[1].Name = "Name";
        dataGridView1.Columns[2].Name = "City";
        dataGridView1.Columns["ID"].HeaderText = "ID";
        dataGridView1.Columns["Name"].HeaderText = "Name";
        dataGridView1.Columns["City"].HeaderText = "City";

        // Add rows of data to the DataGridView.
        dataGridView1.Rows.Add(new string[] { "1", "Parker", "Seattle" });
        dataGridView1.Rows.Add(new string[] { "2", "Parker", "New York" });
        dataGridView1.Rows.Add(new string[] { "3", "Watson", "Seattle" });
        dataGridView1.Rows.Add(new string[] { "4", "Jameson", "New Jersey" });
        dataGridView1.Rows.Add(new string[] { "5", "Brock", "New York" });
        dataGridView1.Rows.Add(new string[] { "6", "Conner", "Portland" });

        // Autosize the columns.
        dataGridView1.AutoResizeColumns();
    }
```

```
    private void dataGridView1_SortCompare(object sender,
        DataGridViewSortCompareEventArgs e)
    {
        // Try to sort based on the cells in the current column.
        e.SortResult = System.String.Compare(
            e.CellValue1.ToString(), e.CellValue2.ToString());

        // If the cells are equal, sort based on the ID column.
        if (e.SortResult == 0 && e.Column.Name != "ID")
        {
            e.SortResult = System.String.Compare(
                dataGridView1.Rows[e.RowIndex1].Cells["ID"].Value.ToString(),
                dataGridView1.Rows[e.RowIndex2].Cells["ID"].Value.ToString());
        }
        e.Handled = true;
    }
}
```

9.2.2 Custom Sorting Using the IComparer Interface

The following code example demonstrates custom sorting using the Sort(IComparer) overload of the Sort method, which takes an implementation of the IComparer interface to perform a multiple-column sort.

```
using System;
using System.Drawing;
using System.Windows.Forms;

class Form1 : Form
{
    private DataGridView DataGridView1 = new DataGridView();
    private FlowLayoutPanel FlowLayoutPanel1 = new FlowLayoutPanel();
    private Button Button1 = new Button();
    private RadioButton RadioButton1 = new RadioButton();
    private RadioButton RadioButton2 = new RadioButton();

    // Establish the main entry point for the application.
    [STAThreadAttribute()]
    public static void Main()
    {
        Application.Run(new Form1());
    }

    public Form1()
    {
        // Initialize the form.
        // This code can be replaced with designer generated code.
        AutoSize = true;
        Text = "DataGridView IComparer sort demo";

        FlowLayoutPanel1.FlowDirection = FlowDirection.TopDown;
        FlowLayoutPanel1.Location = new System.Drawing.Point(304, 0);
        FlowLayoutPanel1.AutoSize = true;

        FlowLayoutPanel1.Controls.Add(RadioButton1);
        FlowLayoutPanel1.Controls.Add(RadioButton2);
        FlowLayoutPanel1.Controls.Add(Button1);

        Button1.Text = "Sort";
        RadioButton1.Text = "Ascending";
        RadioButton2.Text = "Descending";
        RadioButton1.Checked = true;
```



```
        Controls.Add(FlowLayoutPanel1);
        Controls.Add(DataGridView1);
    }

    protected override void OnLoad(EventArgs e)
    {
        PopulateDataGridView();
        Button1.Click += new EventHandler(Button1_Click);

        base.OnLoad(e);
    }

    // Replace this with your own code to populate the DataGridView.
    private void PopulateDataGridView()
    {

        DataGridView1.Size = new Size(300, 300);

        // Add columns to the DataGridView.
        DataGridView1.ColumnCount = 2;

        // Set the properties of the DataGridView columns.
        DataGridView1.Columns[0].Name = "First";
        DataGridView1.Columns[1].Name = "Last";
        DataGridView1.Columns["First"].HeaderText = "First Name";
        DataGridView1.Columns["Last"].HeaderText = "Last Name";
        DataGridView1.Columns["First"].SortMode =
            DataGridViewColumnSortMode.Programmatic;
        DataGridView1.Columns["Last"].SortMode =
            DataGridViewColumnSortMode.Programmatic;

        // Add rows of data to the DataGridView.
        DataGridView1.Rows.Add(new string[] { "Peter", "Parker" });
        DataGridView1.Rows.Add(new string[] { "James", "Jameson" });
        DataGridView1.Rows.Add(new string[] { "May", "Parker" });
        DataGridView1.Rows.Add(new string[] { "Mary", "Watson" });
        DataGridView1.Rows.Add(new string[] { "Eddie", "Brock" });
    }

    private void Button1_Click(object sender, EventArgs e)
    {
        if (RadioButton1.Checked == true)
        {
            DataGridView1.Sort(new RowComparer(SortOrder.Ascending));
        }
        else if (RadioButton2.Checked == true)
        {
            DataGridView1.Sort(new RowComparer(SortOrder.Descending));
        }
    }

    private class RowComparer : System.Collections.IComparer
    {
        private static int sortOrderModifier = 1;

        public RowComparer(SortOrder sortOrder)
        {
            if (sortOrder == SortOrder.Descending)
            {
                sortOrderModifier = -1;
            }
            else if (sortOrder == SortOrder.Ascending)
```

```
            {
                sortOrderModifier = 1;
            }
        }

        public int Compare(object x, object y)
        {
            DataGridViewRow DataGridViewRow1 = (DataGridViewRow)x;
            DataGridViewRow DataGridViewRow2 = (DataGridViewRow)y;

            // Try to sort based on the Last Name column.
            int CompareResult = System.String.Compare(
                DataGridViewRow1.Cells[1].Value.ToString(),
                DataGridViewRow2.Cells[1].Value.ToString());

            // If the Last Names are equal, sort based on the First Name.
            if (CompareResult == 0)
            {
                CompareResult = System.String.Compare(
                    DataGridViewRow1.Cells[0].Value.ToString(),
                    DataGridViewRow2.Cells[0].Value.ToString());
            }
            return CompareResult * sortOrderModifier;
        }
    }
}
```

10. How do I hook up events on the editing control?

Sometimes you will need to handle specific events provided by the editing control for a cell. You can do this by first handling the DataGridView.EditingControlShowing event. Next access the DataGridViewEditingControlShowingEventArgs.Control property to get the editing control for the cell. You might need to cast the control to a specific control type if the event you are interested is not based of the Control class.

NOTE: The DataGridView reuses editing controls across cells if the type is the same. Because of this you should make sure that you do not continuously hook up a new event handler if there is already one hooked up otherwise you your event handler will get called multiple times.

11. When should I remove event handlers from the editing control?

If you hook up an event handler on your editing control that is temporary (maybe for a specific cell in a specific column) you can remove the event handler in the CellEndEdit event. You can also remove any existing event handlers before adding an event handler.

12. How do I handle the SelectedIndexChanged event?

Sometimes it is helpful to know when a user has selected an item in the ComboBox editing control. With a ComboBox on your form you would normally handle the SelectedIndexChanged event. With the DataGridViewComboBox you can do the same thing by using the DataGridView.EditingControlShowing event. The following code example demonstrates how to do this. Note that the sample also demonstrates how to keep multiple SelectedIndexChanged events from firing.

```
private void dataGridView1_EditingControlShowing(object sender,
                    DataGridViewEditingControlShowingEventArgs e)
{
    ComboBox cb = e.Control as ComboBox;
```



```
    if (cb != null)
    {
        // first remove event handler to keep from attaching multiple:
        cb.SelectedIndexChanged -= new
            EventHandler(cb_SelectedIndexChanged);

        // now attach the event handler
        cb.SelectedIndexChanged += new
            EventHandler(cb_SelectedIndexChanged);
    }
}

void cb_SelectedIndexChanged(object sender, EventArgs e)
{
    MessageBox.Show("Selected index changed");
}
```

13. How do I perform drag and drop reorder of rows?

Drag and dropping to reorder rows is not built into the DataGridView, but following standard drag and drop code you can easily add this functionality to the DataGridView. The code fragment below shows how you can accomplish this. It assumes that you have a DataGridView control on your form named dataGridView1 and that the grid's AllowDrop property is true and the necessary events are hooked up to the correct event handlers.

```
private Rectangle dragBoxFromMouseDown;
private int rowIndexFromMouseDown;
private int rowIndexOfItemUnderMouseToDrop;
private void dataGridView1_MouseMove(object sender, MouseEventArgs e)
{
    if ((e.Button & MouseButtons.Left) == MouseButtons.Left)
    {
        // If the mouse moves outside the rectangle, start the drag.
        if (dragBoxFromMouseDown != Rectangle.Empty &&
            !dragBoxFromMouseDown.Contains(e.X, e.Y))
        {

            // Proceed with the drag and drop, passing in the list item.
            DragDropEffects dropEffect = dataGridView1.DoDragDrop(
                    dataGridView1.Rows[rowIndexFromMouseDown],
                    DragDropEffects.Move);
        }
    }
}

private void dataGridView1_MouseDown(object sender, MouseEventArgs e)
{
    // Get the index of the item the mouse is below.
    rowIndexFromMouseDown = dataGridView1.HitTest(e.X, e.Y).RowIndex;

    if (rowIndexFromMouseDown != -1)
    {
        // Remember the point where the mouse down occurred.
        // The DragSize indicates the size that the mouse can move
        // before a drag event should be started.
        Size dragSize = SystemInformation.DragSize;

        // Create a rectangle using the DragSize, with the mouse position being
        // at the center of the rectangle.
        dragBoxFromMouseDown = new Rectangle(new Point(e.X - (dragSize.Width / 2),
```

```
                                                e.Y - (dragSize.Height / 2)),
                                        dragSize);
    }
    else
        // Reset the rectangle if the mouse is not over an item in the ListBox.
        dragBoxFromMouseDown = Rectangle.Empty;
}

private void dataGridView1_DragOver(object sender, DragEventArgs e)
{
    e.Effect = DragDropEffects.Move;
}

private void dataGridView1_DragDrop(object sender, DragEventArgs e)
{
    // The mouse locations are relative to the screen, so they must be
    // converted to client coordinates.
    Point clientPoint = dataGridView1.PointToClient(new Point(e.X, e.Y));

    // Get the row index of the item the mouse is below.
    rowIndexOfItemUnderMouseToDrop =
        dataGridView1.HitTest(clientPoint.X, clientPoint.Y).RowIndex;

    // If the drag operation was a move then remove and insert the row.
    if (e.Effect== DragDropEffects.Move)
    {
        DataGridViewRow rowToMove = e.Data.GetData(
                    typeof(DataGridViewRow)) as DataGridViewRow;
        dataGridView1.Rows.RemoveAt(rowIndexFromMouseDown);
        dataGridView1.Rows.Insert(rowIndexOfItemUnderMouseToDrop, rowToMove);

    }
}
```

14. How do I make the last column wide enough to occupy all the remaining client area of the grid?

By setting the AutoSizeMode for the last column to Fill the column will size itself to fill in the remaining client area of the grid. Optionally you can set the last column's MinimumWidth if you want to keep the column from sizing too small.

15. How do I have the cell text wrap?

By default, text in a DataGridViewTextBoxCell does not wrap. This can be controlled via the WrapMode property on the cell style (e.g. DataGridView.DefaultCellStyle.WrapMode). Set the WrapMode property of a DataGridViewCellStyle to one of the DataGridViewTriState enumeration values.

The following code example uses the DataGridView.DefaultCellStyle property to set the wrap mode for the entire control.

```
this.dataGridView1.DefaultCellStyle.WrapMode = DataGridViewTriState.True;
```

16. How do I make the image column not show any images?

By default the image column and cell convert null values to the standard "X" image (✕). You can make no image show up by changing the column's NullValue property to null. The following code example sets the NullValue for an image column:


```
this.dataGridViewImageColumn1.DefaultCellStyle.NullValue = null;
```

17. How do I enable typing in the combo box cell?

By default a DataGridViewComboBoxCell does not support typing into the cell. There are reasons though that typing into the combo box works well for your application. To enable this, two things have to be done. First the DropDownStyle property of the ComboBox editing control needs to be set to DropDown to enable typing in the combo box. The second thing that needs to be done is to ensure that the value that the user typed into the cell is added to the combo box items collection. This is due to the requirement described in 3.5.1 section that a combo box cells value must be in the items collection or else a DataError event is raised. The appropriate place to add the value to the items collection is in the CellValidating event handler.

```
private void dataGridView1_CellValidating(object sender,
                DataGridViewCellValidatingEventArgs e)
{
    if (e.ColumnIndex == comboBoxColumn.DisplayIndex)
    {
        if (!this.comboBoxColumn.Items.Contains(e.FormattedValue))
        {
            this.comboBoxColumn.Items.Add(e.FormattedValue);
        }
    }
}

private void dataGridView1_EditingControlShowing(object sender,
              DataGridViewEditingControlShowingEventArgs e)
{
    if (this.dataGridView1.CurrentCellAddress.X == comboBoxColumn.DisplayIndex)
    {
        ComboBox cb = e.Control as ComboBox;
        if (cb != null)
        {
            cb.DropDownStyle = ComboBoxStyle.DropDown;
         }
    }
}
```

18. How do I have a combo box column display a sub set of data based upon the value of a different combo box column?

Sometimes data that you want to display in the DataGridView has a relationship between two tables such as a category and subcategory. You want to let the user select the category and then choose between a subcategory based upon the category. This is possible with the DataGridView by using two combo box columns. To enable this, two versions of the filtered list (subcategory) needs to be created. One list has no filter applied while the other one will be filtered only when the user is editing a subcategory cell. Two lists are required due to the requirement described in 3.5.1 section that a combo box cells value must be in the items collection or else a DataError event is raised. In this case, since all combo box cells in the column use the same datasource if you filter the datasource for one row then a combo box cell in another row might not have its value visible in the datasource, thus causing a DataError event.

The below example uses the Northwind database to display related data from the Territory and Region tables (a territory is in a specific region.) Using the category and subcategory concept, the Region is the category and the Territory is the subcategory.

```
private void Form1_Load(object sender, EventArgs e)
{
    this.territoriesTableAdapter.Fill(this.northwindDataSet.Territories);
    this.regionTableAdapter.Fill(this.northwindDataSet.Region);

    // Setup BindingSource for filtered view.
    filteredTerritoriesBS = new BindingSource();
    DataView dv = new DataView(northwindDataSet.Tables["Territories"]);
    filteredTerritoriesBS.DataSource = dv;

}

private void dataGridView1_CellBeginEdit(object sender,
             DataGridViewCellCancelEventArgs e)
{
    if (e.ColumnIndex == territoryComboBoxColumn.Index)
    {
        // Set the combobox cell datasource to the filtered BindingSource
        DataGridViewComboBoxCell dgcb = (DataGridViewComboBoxCell)dataGridView1
                                        [e.ColumnIndex, e.RowIndex];
        dgcb.DataSource = filteredTerritoriesBS;

        // Filter the BindingSource based upon the region selected
        this.filteredTerritoriesBS.Filter = "RegionID = " +
            this.dataGridView1[e.ColumnIndex - 1, e.RowIndex].Value.ToString();
    }
}

private void dataGridView1_CellEndEdit(object sender, DataGridViewCellEventArgs e)
{
    if (e.ColumnIndex == this.territoryComboBoxColumn.Index)
    {
        // Reset combobox cell to the unfiltered BindingSource
        DataGridViewComboBoxCell dgcb = (DataGridViewComboBoxCell)dataGridView1
                                        [e.ColumnIndex, e.RowIndex];
        dgcb.DataSource = territoriesBindingSource; //unfiltered

        this.filteredTerritoriesBS.RemoveFilter();
    }
}
```

19. How do I show the error icon when the user is editing the cell?

Sometimes when using the error text and icon feature you want an immediate feedback to the user that something that they typed into a cell is incorrect. By default when setting the ErrorText property the error icon will not appear if the cell is in edit mode such as a text box or combo box cell.

The below sample demonstrates how you can set a cell's padding in the CellValidating event to provide spacing for the error icon. Since padding by default affects the location of the error icon the sample uses the CellPainting to move the position of the icon for painting. Lastly, the sample uses the tooltip control to display a custom tooltip when the mouse is over the cell to indicate what the problem is. This sample could also be written as a custom cell that overrides GetErrorIconBounds method to provide a location for the error icon that was independent of the padding.

```
private ToolTip errorTooltip;
private Point cellInError = new Point(-2, -2);
public Form1()
{
    InitializeComponent();
```



```
    dataGridView1.ColumnCount = 3;
    dataGridView1.RowCount = 10;
}

private void dataGridView1_CellValidating(object sender,
DataGridViewCellValidatingEventArgs e)
{
    if (dataGridView1.IsCurrentCellDirty)
    {
        if (e.FormattedValue.ToString() == "BAD")
        {
            DataGridViewCell cell = dataGridView1[e.ColumnIndex, e.RowIndex];
            cell.ErrorText = "Invalid data entered in cell";

            // increase padding for icon. This moves the editing control
            if (cell.Tag == null)
            {
                cell.Tag = cell.Style.Padding;
                cell.Style.Padding = new Padding(0, 0, 18, 0);
                cellInError = new Point(e.ColumnIndex, e.RowIndex);
            }
            if (errorTooltip == null)
            {
                errorTooltip = new ToolTip();
                errorTooltip.InitialDelay = 0;
                errorTooltip.ReshowDelay = 0;
                errorTooltip.Active = false;
            }

            e.Cancel = true;
        }
    }
}

private void dataGridView1_CellPainting(object sender,
DataGridViewCellPaintingEventArgs e)
{
    if (dataGridView1.IsCurrentCellDirty && !String.IsNullOrEmpty(e.ErrorText))
    {
        // paint everything except error icon
        e.Paint(e.ClipBounds, DataGridViewPaintParts.All &
                    ~(DataGridViewPaintParts.ErrorIcon));

        // now move error icon over to fill in the padding space
        GraphicsContainer container = e.Graphics.BeginContainer();
        e.Graphics.TranslateTransform(18, 0);
        e.Paint(this.ClientRectangle, DataGridViewPaintParts.ErrorIcon);
        e.Graphics.EndContainer(container);

        e.Handled = true;
    }
}

private void dataGridView1_CellEndEdit(object sender, DataGridViewCellEventArgs e)
{
    if (dataGridView1[e.ColumnIndex, e.RowIndex].ErrorText != String.Empty)
    {
        DataGridViewCell cell = dataGridView1[e.ColumnIndex, e.RowIndex];
        cell.ErrorText = String.Empty;
        cellInError = new Point(-2,-2);

        // restore padding for cell. This moves the editing control
        cell.Style.Padding = (Padding)cell.Tag;
```

```
        // hide and dispose tooltip
        if (errorTooltip != null)
        {
            errorTooltip.Hide(dataGridView1);
            errorTooltip.Dispose();
            errorTooltip = null;
        }
    }
}

// show and hide the tooltip for error
private void dataGridView1_CellMouseMove(object sender,
                    DataGridViewCellMouseEventArgs e)
{
    if (cellInError.X == e.ColumnIndex &&
        cellInError.Y == e.RowIndex)
    {
        DataGridViewCell cell = dataGridView1[e.ColumnIndex, e.RowIndex];

        if (cell.ErrorText != String.Empty)
        {
            if (!errorTooltip.Active)
            {
                errorTooltip.Show(cell.ErrorText, dataGridView1, 1000);
            }
            errorTooltip.Active = true;
        }
    }
}

private void dataGridView1_CellMouseLeave(object sender, DataGridViewCellEventArgs e)
{
    if (cellInError.X == e.ColumnIndex &&
        cellInError.Y == e.RowIndex)
    {
        if (errorTooltip.Active)
        {
            errorTooltip.Hide(dataGridView1);
            errorTooltip.Active = false;
        }
    }
}
```

20. How do I show unbound data along with bound data?

The data you display in the DataGridView control will normally come from a data source of some kind, but you might want to display a column of data that does not come from the data source. This kind of column is called an unbound column. Unbound columns can take many forms. As discussed in the data section above, you can use virtual mode to display additional data along with bound data.

The following code example demonstrates how to create an unbound column of check box cells to enable the user to select database records to process. The grid is put into virtual mode and responds to the necessary events. The selected records are kept by ID in a dictionary to allow the user to sort the content but not lose the checked rows.

```
private System.Collections.Generic.Dictionary<int, bool> checkState;
private void Form1_Load(object sender, EventArgs e)
{
    dataGridView1.AutoGenerateColumns = false;
    dataGridView1.DataSource = customerOrdersBindingSource;
```



```
    // The check box column will be virtual.
    dataGridView1.VirtualMode = true;
    dataGridView1.Columns.Insert(0, new DataGridViewCheckBoxColumn());

    // Initialize the dictionary that contains the boolean check state.
    checkState = new Dictionary<int, bool>();
}
private void dataGridView1_CellValueChanged(object sender, DataGridViewCellEventArgs
e)
{
    // Update the status bar when the cell value changes.
    if (e.ColumnIndex == 0 && e.RowIndex != -1)
    {
        // Get the orderID from the OrderID column.
        int orderID = (int)dataGridView1.Rows[e.RowIndex].Cells["OrderID"].Value;
        checkState[orderID] = (bool)dataGridView1.Rows[e.RowIndex].Cells[0].Value;

}

private void dataGridView1_CellValueNeeded(object sender,
DataGridViewCellValueEventArgs e)
{
    // Handle the notification that the value for a cell in the virtual column
    // is needed. Get the value from the dictionary if the key exists.

    if (e.ColumnIndex == 0)
    {
        int orderID = (int)dataGridView1.Rows[e.RowIndex].Cells["OrderID"].Value;
        if (checkState.ContainsKey(orderID))
        {
            e.Value = checkState[orderID];
        }
        else
            e.Value = false;
    }

}

private void dataGridView1_CellValuePushed(object sender,
DataGridViewCellValueEventArgs e)
{
    // Handle the notification that the value for a cell in the virtual column
    // needs to be pushed back to the dictionary.

    if (e.ColumnIndex == 0)
    {
        // Get the orderID from the OrderID column.
        int orderID = (int)dataGridView1.Rows[e.RowIndex].Cells["OrderID"].Value;

        // Add or update the checked value to the dictionary depending on if the
        // key (orderID) already exists.
        if (!checkState.ContainsKey(orderID))
        {
            checkState.Add(orderID, (bool)e.Value);
        }
        else
            checkState[orderID] = (bool)e.Value;
    }
}
```

21. How do I show data that comes from two tables?

The DataGridView does not provide any new features apart from virtual mode to enable this. What you can do is use the JoinView class described in the following article http://support.microsoft.com/default.aspx?scid=kb;en-us;325682. Using this class you can join two or more DataTables together. This JoinView can then be databound to the DataGridView.

22. How do I show master-details?

One of the most common scenarios for using the DataGridView control is the master/detail form, in which a parent/child relationship between two database tables is displayed. Selecting rows in the master table causes the detail table to update with the corresponding child data.

Implementing a master/detail form is easy using the interaction between the DataGridView control and the BindingSource component. The below sample will show two related tables in the Northwind SQL Server sample database: Customers and Orders. By selecting a customer in the master DataGridView all the orders for the customer will appear in the detail DataGridView.

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Windows.Forms;

public class Form1 : System.Windows.Forms.Form
{
    private DataGridView masterDataGridView = new DataGridView();
    private BindingSource masterBindingSource = new BindingSource();
    private DataGridView detailsDataGridView = new DataGridView();
    private BindingSource detailsBindingSource = new BindingSource();

    [STAThreadAttribute()]
    public static void Main()
    {
        Application.Run(new Form1());
    }

    // Initializes the form.
    public Form1()
    {
        masterDataGridView.Dock = DockStyle.Fill;
        detailsDataGridView.Dock = DockStyle.Fill;

        SplitContainer splitContainer1 = new SplitContainer();
        splitContainer1.Dock = DockStyle.Fill;
        splitContainer1.Orientation = Orientation.Horizontal;
        splitContainer1.Panel1.Controls.Add(masterDataGridView);
        splitContainer1.Panel2.Controls.Add(detailsDataGridView);

        this.Controls.Add(splitContainer1);
        this.Load += new System.EventHandler(Form1_Load);
        this.Text = "DataGridView master/detail demo";
    }

    private void Form1_Load(object sender, System.EventArgs e)
    {
        // Bind the DataGridView controls to the BindingSource
        // components and load the data from the database.
        masterDataGridView.DataSource = masterBindingSource;
        detailsDataGridView.DataSource = detailsBindingSource;
        GetData();
```



```
        // Resize the master DataGridView columns to fit the newly loaded data.
        masterDataGridView.AutoResizeColumns();

        // Configure the details DataGridView so that its columns automatically
        // adjust their widths when the data changes.
        detailsDataGridView.AutoSizeColumnsMode =
            DataGridViewAutoSizeColumnsMode.AllCells;
    }

    private void GetData()
    {
        try
        {
            // Specify a connection string. Replace the given value with a
            // valid connection string for a Northwind SQL Server sample
            // database accessible to your system.
            String connectionString =
                "Integrated Security=SSPI;Persist Security Info=False;" +
                "Initial Catalog=Northwind;Data Source=localhost";
            SqlConnection connection = new SqlConnection(connectionString);

            // Create a DataSet.
            DataSet data = new DataSet();
            data.Locale = System.Globalization.CultureInfo.InvariantCulture;

            // Add data from the Customers table to the DataSet.
            SqlDataAdapter masterDataAdapter = new
                SqlDataAdapter("select * from Customers", connection);
            masterDataAdapter.Fill(data, "Customers");

            // Add data from the Orders table to the DataSet.
            SqlDataAdapter detailsDataAdapter = new
                SqlDataAdapter("select * from Orders", connection);
            detailsDataAdapter.Fill(data, "Orders");

            // Establish a relationship between the two tables.
            DataRelation relation = new DataRelation("CustomersOrders",
                data.Tables["Customers"].Columns["CustomerID"],
                data.Tables["Orders"].Columns["CustomerID"]);
            data.Relations.Add(relation);

            // Bind the master data connector to the Customers table.
            masterBindingSource.DataSource = data;
            masterBindingSource.DataMember = "Customers";

            // Bind the details data connector to the master data connector,
            // using the DataRelation name to filter the information in the
            // details table based on the current row in the master table.
            detailsBindingSource.DataSource = masterBindingSource;
            detailsBindingSource.DataMember = "CustomersOrders";
        }
        catch (SqlException)
        {
            MessageBox.Show("To run this example, replace the value of the " +
                "connectionString variable with a connection string that is " +
                "valid for your system.");
        }
    }
}
```

23. How do I show master-details in the same DataGridView?

The DataGridView does not provide any support for showing master-details data in the same DataGridView. The previously shipped Windows Forms DataGrid control can be a solution if this is something that you need.

24. How do I prevent sorting?

By setting the DataGridViewColumn.SortMode property you can disable the ability for the user to sort on the given column. You can use Visual Studio 2005 to set this property by right-clicking on the DataGridView and choosing the Edit Columns option. Next select the column that you want to disable sorting for and set the SortMode property to NotSortable.

25. How do I commit the data to the database when clicking on a toolstrip button?

By default, toolbars and menus do not force validation for controls. Validation is a required part of validating and updating the data in a bound control. Once the form and all bound controls are validated, any current edits need to be committed. Finally, the table adapter needs to push its changes back to the database. To do this, put the following three lines in a click event handler on your form:

```
this.Validate();
this.customersBindingSource.EndEdit();
this.customersTableAdapter.Update(this.northwindDataSet.Customers);
```

26. How do I display a confirmation dialog when the user tries to delete a row?

When the user selects a row in the DataGridView and hits the delete key, the UserDeletingRow event fires. You can prompt the user if they want to continue deleting the row. It is recommended that you only do this if the row being deleted is not the new row. Add the following code to the UserDeletingRow event handler to perform this:

```
if (!e.Row.IsNewRow)
{
    DialogResult response = MessageBox.Show("Are you sure?", "Delete row?",
                      MessageBoxButtons.YesNo,
                      MessageBoxIcon.Question,
                      MessageBoxDefaultButton.Button2);
    if (response == DialogResult.No)
        e.Cancel = true;
}
```