OBJECTIVE GRID FOR MICROSOFT® .NET® USER'S GUIDE

Version 11.3.0

ROGUE WAVE SOFTWARE / 5500 FLATIRON PARKWAY, SUITE 200 / BOULDER, CO 80301, USA / **WWW.ROGUEWAVE.COM**

OBJECTIVE GRID FOR MICROSOFT .NET USER'S GUIDE

THIS MANUAL

© Copyright 1997-2015 Rogue Wave Software, Inc. All Rights Reserved.

Rogue Wave and Stingray are registered trademarks of Rogue Wave Software, Inc. in the United States and other countries. All other trademarks are the property of their respective owners.

Contents

Chapter 1 **Introduction to Objective Grid for Microsoft® .NET®**

Welcome to Objective Grid for Microsoft .NET 1
Product Features 1
Supported Platforms 2
Distributing Objective Grid for Microsoft .NET Applications 3
Naming Conventions 3
Using This Manual 4
Common Terms 4
Getting Help 5
Documentation 5
Tutorials and Samples 6
Knowledge Base 7
Professional Services 7
Technical Support 7

Chapter 2 **Design Overview**

Introduction 8
Description of the Core Classes 9
The Cell Class 10
The Range Class 12
The Style Class 15
The Param Class 19
The GridControl Class 21
Predefined Controls 35
Browser Architecture 37
Formula Engine Support 37
Enabling Formula Support 37
Mathematical Functions 38

Statistical Functions. 40
Conditional Statistical Functions 42
String Functions 43
Logic Functions 44
Financial Functions 45
Date and Time Functions. 48
Miscellaneous Functions 49
Embedded Tools 50
Choosing a Grid Type 52

Chapter 3 Using Objective Grid for Microsoft .NET

Introduction 54
Basic Grid Tasks. 55
Task 1: Changing the Number of Rows and Columns in the Grid 55
Task 2: Changing the Number of Rows and Columns after Initialization 56
Task 3: Storing Data in the Grid 57
Task 4: Reading Data from the Grid 58
Cell Types 60
Task 1: Modify the Attribute of a Cell by Embedding a Control 61
Task 2: Disabling Cells. 62
Task 3: Making a Cell Readonly. 63
Task 4: Covering Cells 63
Task 5: Merging Cells 63
Columns and Rows 65
Task 1: Inserting New Rows and Columns 66
Task 2: Removing Rows and Columns 67
Task 3: Hiding Rows and Columns. 67
Task 4: Freezing Rows and Columns. 69
Cell and Window Coordinates 71
Sorting a Grid 74
Extending .NET Functionality 76
Customizing Behavior. 76
Customizing the Display. 83
Printing. 90
Sorting 91
Memory and Performance Issues 92
Troubleshooting. 94

Chapter 4 Virtual Grids

Background 95
Reasons for Using Virtual Grids 96
Using Virtual Grids 97
Advanced Virtual Grid Topics 98

Chapter 5 .NET Cell Controls

Introduction 99
Overview 99
Features 101
.NET Controls Can Be Used as Cell Editors 101
Memory Resources Are Conserved 101
Style Manipulation Code Is Usually Not Needed 101
Cell Control Behavior Can Be Monitored and Customized 101
Cell Editing Events Can Be Monitored 102
Limitations 103
Some Controls Cannot Be Sized Vertically 103
CellControl Behavior Must Be Customized Using Delegates 103
The .NET Control Can Be Drawn in Only One Cell at a Time 103
Supports Registration of a Maximum of 32 .NET Controls 103
Using .NET Cell Controls 104

Chapter 6 GridTabControl Control

Introduction 107
Overview 108
Structure and Design 109
Design Time Support 110
Using GridTabControl 111

Chapter 7 Hierarchical Grids

Introduction 112
Hierarchical Grid Methods and Properties 114
Using Hierarchical Grids 116
Advanced Topics 118
The HgMap Property 118
Using the HgMap Property 119

Chapter 8 1stGrid Tutorial

Introduction 120

1stGrid - Step 1 122

- Create a New Solution and Project 122
- Add a GridControl 123

1stGrid - Step 2 125

- Create a New Project 125
- Add a Menu 125
- Add the GridDocument Class 129
- Add an “About” Box 131
- Add MDI Support 132
- Add Event Handlers 133

1stGrid - Step 3 136

- Create a New Project 136
- Add a Property Grid 136

1stGrid - Step 4 141

- Create a New Project 141
- Enable the Formula Engine 141
- Add Event Handlers 142

Chapter 9 Samples

Introduction 144

SimpleSpreadSheet 146

ExcelLike 147

- Running the Sample 147
- Formula Support 148
- Copying and Pasting Data and Formulas 149
- Undo and Redo Changes 151
- Find, Find and Replace 151
- Saving a Grid in HTML Format 153
- Design-time Features 154

Virtual Grid 156

- Create a solution for the virtual grid project 156
- Place a GridControl on Form1.cs. 156
- Create a Data Structure for Storing Grid Cell Values 156
- Add Virtual Grid Support to the Project 161
- Add Event Handlers for the Menu Commands and Virtual Grid Events 162

RaceAttendance 164

- Connecting the Property Grid to the Grid 165

Adding Data to the Grid 165
Formulas 166
Controls and Event Handling 167
Miscellaneous 168

Chapter 1 **Working with the Office Open XML 2.5 SDK**

Office Open XML SDK for Excel 1
OOXML Excel Library Addition 3
OOXML Excel Library Example 5

Index 7

Chapter 1

Introduction to Objective Grid for Microsoft® .NET®

Welcome to Objective Grid for Microsoft .NET

Objective Grid for Microsoft .NET is an object-oriented .NET grid control, a user interface component that displays data in rows and columns. Objective Grid for Microsoft .NET includes:

- Support for embedding grid objects in Windows Forms
- A set of pre-defined cell controls that can be embedded in grid cells
- The ability to embed any .NET control in a grid cell

Objective Grid for Microsoft .NET is compatible with the latest releases of Microsoft Visual Studio .NET.

Product Features

The features of Objective Grid for Microsoft .NET include:

- **Many built-in cell types**—Objective Grid for Microsoft .NET offers more than 30 cell types, ranging from simple edit controls to formatted date-time and currency controls.
- **Ability to embed your own controls**—If one of the built-in controls does not fit your needs, you can embed your own control.

- **Database connectivity**—Objective Grid for Microsoft .NET ships with built-in support for ADO.NET data sources.
- **Undo/Redo**—Objective Grid for Microsoft .NET fully supports Undo/Redo for built-in grid operations, including transactions and rollbacks. The architecture can be easily extended to include application-specific operations.
- **Printing and print preview**—Objective Grid for Microsoft .NET includes full print and print preview support.
- **Find/Replace**—Objective Grid for Microsoft .NET includes support for finding and replacing cell data.
- **Cut/Copy/Paste**—Objective Grid for Microsoft .NET supports an internal direct cut/copy/paste and cut/copy/paste using the Windows clipboard.
- **Object-oriented cell architecture**—Objective Grid for Microsoft .NET uses a control sharing architecture in which each type of cell shares a single control with all similar types. Cell data is stored separately from the cell control. This system saves space and resources.
- **Excel-like interface**—Objective Grid for Microsoft .NET enables the programmer to mimic many of the user interface features in Microsoft Excel.
- **Tab Grid**—Tabbed control that allows for creation of interfaces similar to Excel tabbed worksheets.
- **Floating and merged cells**—Objective Grid for Microsoft .NET includes support for both floating and merged cells. Floating cells are cells that automatically grow over adjacent cells to accommodate text too large for a single cell. Merged cells are adjacent cells that contain the same value and are drawn as a single large cell.
- **Formula support**—The Objective Grid for Microsoft .NET formula engine has more than 200 built-in worksheet functions and can be extended with your own custom worksheet functions.

Supported Platforms

For a list of supported operating systems and compilers, see `http://www.roguewave.com/products/stingray.aspx`, then click on the link “Supported Platforms” to download a PDF.

Distributing Objective Grid for Microsoft .NET Applications

Please read the license agreement that was shipped with this package. You are bound by the licensing restrictions contained in that document. Do not use this product unless you can accept all the terms of the license agreement.

You can use all the files accompanying this product for development of an application. You can distribute the Objective Grid for Microsoft .NET assembly according to the terms of the license agreement. Your applications can also statically link to the grid, in which case you do not need to redistribute any Objective Grid for Microsoft .NET files.

Naming Conventions

Objective Grid for Microsoft .NET includes both release and debug versions of the Objective Grid for Microsoft .NET assembly. Their names follow Microsoft's standard for assembly names.

Table 1 – Objective Grid for Microsoft .NET Assemblies

Assembly	File
Stingray.Grid	`Stingray.GridUtils.dll`
Stingray.GridControl	`Stingray.GridControl.dll`
Stingray.GridTabControl	`Stingray.GridTabControl.dll`

Using This Manual

Throughout this document, a set of typographical conventions are used to define elements and references to Objective Grid for Microsoft .NET items. Familiarity with these conventions will help your understanding of the topics covered.

Table 2 – Conventions used in this document

Example	Description
`void myfunc()`	Library routines have parenthesis '()' as a suffix.
Expression	Words in italics indicate placeholders for information you must supply, such as a variable or parameters for methods.
`using Stingray.Grid;`	Courier font is used for code examples.

`while(){`
`  ...`
`}`
	A column or row of three dots (also known as an ellipsis) indicates that part of an example program has been intentionally omitted.

`CTRL+ENTER` `ENTER`	Words in small caps indicate names of keys on the keyboard. A plus sign (+) between two key names indicates that you should hold down the first key while pressing the second.
Menu|Menu Item	Bold font is used for menu commands.

Common Terms

Base Style - Base styles are grid-wide styles that make it possible to group specific kinds of cells and give them similar attributes. The predefined base styles are: row-header-style, column-header-style and standard-style. Row header cells inherit their attributes from row-header-style. Column headers inherit from column-header-style. Standard-style is the base style for all cells in the grid.

Cell - Cells display information in the grid. Each cell has a unique coordinate (row, column). Cells are associated with a control and a style object.

Control - Controls handle the interface between the user and the grid. Each cell is associated with a control. The control interprets user events and is responsible for drawing the cell.

Covered Cells - Covered cells span several other cells and can be used for headings in reports.

Current Cell—The current cell is managed by the grid as the user navigates through the grid by clicking or using arrow keys. The current cell lets the user modify the cell's contents through its associated control.

Data source—Data source is a general term that can mean an ADO.NET query result, a database, or any other external data structure or medium.

Properties—Properties are settings in the grid that can be modified with pre-built dialogs. Properties are maintained by the ***Properties*** class.

Range—A range defines a rectangular area of cells in the grid. A range is specified through a top and bottom row, and left and right columns. Ranges can represent a selection of cells, columns, rows, or all cells.

Style—A style contains all the information necessary for formatting a cell. A style consists of several attributes such as text color, borders, control type, and font. Each cell determines the style information at run time and passes this information to the control for drawing the cell.

Workbook—A workbook lets the user switch between several views connected to the same document by clicking on a tab at the bottom-left of the window.

Worksheet—Worksheet is used to refer to each of the individual views displayed in a workbook.

Getting Help

Several avenues of help are available to you when working with Objective Grid for Microsoft .NET.

Documentation

The documentation for Objective Grid for Microsoft .NET includes:

- ***User's Guide—***This manual. The ***User's Guide*** is an introductory manual for Objective Grid for Microsoft .NET. It includes tutorials to help new users learn how to create Objective Grid for Microsoft .NET applications quickly.
- ***Reference Guide***—The reference guide is a detailed description of the classes and methods in Objective Grid for Microsoft .NET.

Available Formats

The following is a list of available documentation formats. After each format is a list of the documents available in that format.

- **Portable Document Format (PDF) documents—**located in the `Docs` subdirectory of your Objective Grid for Microsoft .NET directory
  - ***User's Guide***—`Objective_Grid_for_Microsoft_.NET_User_Guide.pdf`
- **HTML help—**located in the `Docs` subdirectory of your Objective Grid for Microsoft .NET directory
  - ***User's Guide***—`ognetug.chm`
  - ***Reference Guide***—`ognref.chm`

— ***FormulaEngine***—`FormulaEngine.chm`; also included in the ***Reference Guide***

For more information on the documentation, including all Stingray documentation, an index to the Help files, and document type conventions, see Section 1.4, "Product Documentation," in the *Stingray Studio Getting Started Guide*.

Tutorials and Samples

- **Tutorials**—Tutorials are located in the tutorials subdirectory of your Objective Grid for Microsoft .NET installation directory. They include:
  - 1stGrid—Shows how to use Objective Grid for Microsoft .NET to create a simple grid application.
  - CellValue_CS—Shows how to write to the grid and read from the grid.
  - CellAttribute_CS—Shows how to use styles, covered cells, floating cells, and merged cells.
  - RowsColumns_CS—Shows how to hide and freeze rows and columns.
  - CellWindowCoord_CS—Shows how to use cell and window coordinates.
  - Formula_CS—Shows how to enable formula support in grids.
  - Sort_CS—Shows how to support sorting when users double-click row or column headers.
- **Samples**—Samples are located in the Samples subdirectory of your Objective Grid for Microsoft .NET installation directory. They include:
  - SimpleSpreadSheet—Shows basic spreadsheet functionality.
  - ExcelLike—Demonstrates Objective Grid for Microsoft .NET features that mimic some of the behaviors of Microsoft Excel.
  - Virtual Grid—Shows how to store and retrieve grid data from a data structure.
  - RaceAttendance—Demonstrates properties of the ***Stingray.Grid.Style*** class, and demonstrates printing, print preview, saving grid data to an HTML file, event handling, and embedding a properties sheet so that the grid and cell properties can be manipulated at runtime.
  - Simple .NET Control—Shows the most basic use-case for embedded .NET controls: using a .NET `NumericUpDown` control as a cell editor.
  - Multiple .NET Controls—Shows how to use several different .NET controls as cell editors, and shows how to make use of CellControl properties and events to customize the behavior of the .NET controls.

— Custom .NET Control—Shows how to create a custom .NET control consisting of a `TextBox` with a button on the right of the text, and shows how to use the custom .NET control as a grid cell editor.

— EventHandlingByDelegation—Shows how to handle grid events using delegates.

— EventHandlingByInheritance—Shows how to handle grid events by deriving from GridControl and overriding virtual event handlers.

— Print—Shows how to utilize the print and print preview functionality in the grid.

Knowledge Base

The Rogue Wave Knowledge Base contains a large body of useful information created by the Support Services team. This information is available to any user of the Rogue Wave Web site, and no login or registration is required.
`http://www.roguewave.com/support/knowledge-base.aspx`.

Professional Services

The Rogue Wave Professional Services offers training and mentoring for all levels of project development, from analysis and design to implementation. For more information, see "Professional Services" in the ***Getting Started*** part.

Technical Support

Technical support for Objective Grid for Microsoft .NET products is provided through the Rogue Wave Web site. For more information on registering with the support system, and the type of support you may receive, see ***"Technical Support,"*** in the ***Getting Started*** part.

Chapter 2

Design Overview

Introduction

This chapter describes the following:

- The most commonly used, core classes in the Objective Grid for Microsoft .NET programmer's interface
- The pre-defined controls that can be used in grid cells
- The Browser Grid architecture
- The Objective Grid for Microsoft .NET Formula Engine
- Tips on choosing the grid type appropriate for a given application

More detailed information about each of the interfaces described in this chapter is available in the ***Objective Grid for Microsoft® .NET® Reference Guide***.

Description of the Core Classes

This section describes the core classes in the public interface of the Objective Grid for Microsoft .NET control. All of the classes are contained in the `Stingray.Grid` namespace.

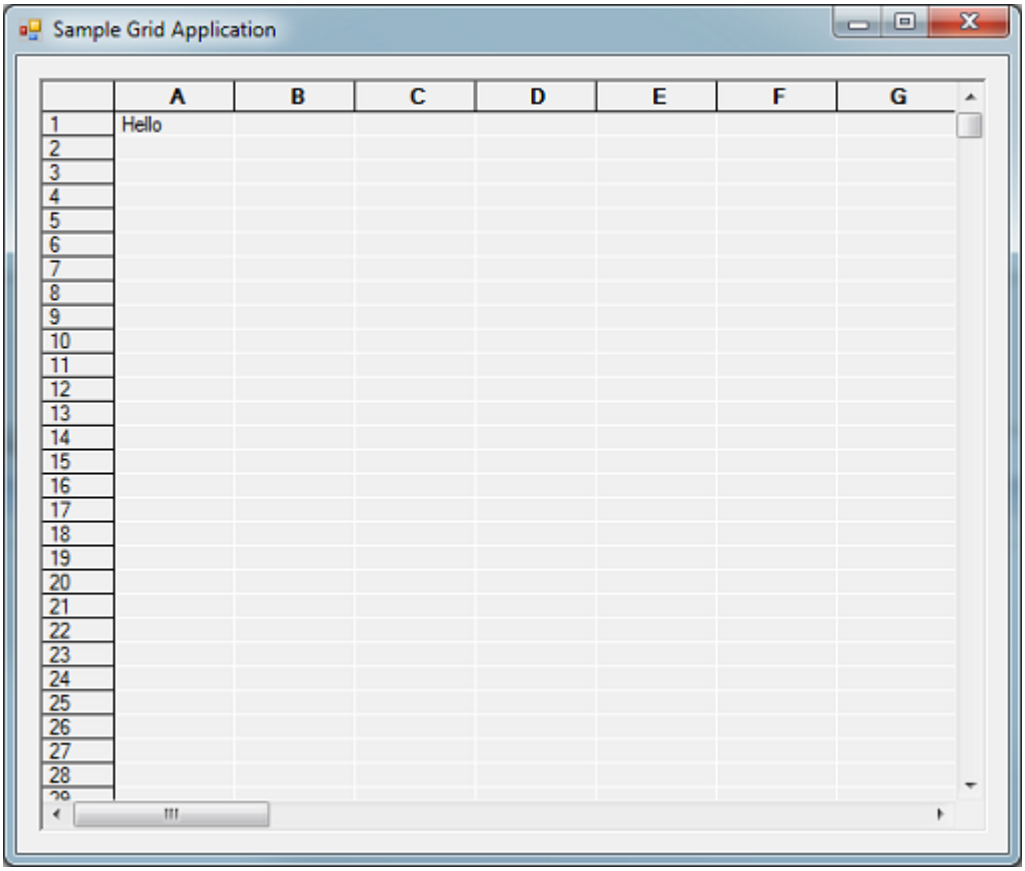
Table 3 – Most Commonly Used Classes

GridControl	The ***GridControl*** class represents and embodies an Objective Grid for Microsoft .NET grid control. The ***GridControl*** custom control is added to the Visual Studio .NET toolbox when Objective Grid for Microsoft .NET is installed. When a ***GridControl*** is dragged onto a Windows Form, an instance of the ***GridControl*** class is created. The ***GridControl*** class contains a rich public interface, a rich set of public events, a number of properties, and a customizable virtual method interface.
Cell	The ***Cell*** class represents a single cell in the grid. A cell is referenced by a row offset and a column offset. The ***Cell*** class contains properties for the cell's row, column, text, and style.
Range	The ***Range*** class represents a range of cells in the grid. The ***Range*** class can be used for operations that apply to a group of cells.
Style	The ***Style*** class is central to formatting cells in the grid. A style object can be seen as an object that completely defines a cell. This means that the style object has all the information necessary for the grid object to draw the cell and manage its interaction with users. This information includes the font used to draw the cell, the color of the cell's interior, the size of the font, the value displayed, and the type of control in the cell. Style objects for each cell are stored internally by the grid. The format of a grid cell can be changed programmatically through the manipulation of the style property of the cell.
Param	The ***Param*** (short for Parameter) class stores all data necessary for persisting the state of a grid object. Parameter objects can be stored in documents and serialized, or used as members of other classes. Parameter objects can also be shared between grid objects.

The Cell Class

The ***Cell*** class represents a single cell in the grid control. A cell is referenced using row and column offsets into the grid. By default, the grid reserves the first row for column headers and the first column for row headers, as shown in Figure 1. The header row and column are numbered row 0 and column 0, respectively.

Figure 1 – GridControl Column and Row Headers



In Figure 1, the cell at row 1 and column A has row and column offsets 1, 1. This cell is referenced in source code as follows:

C#:

```
Cell cell = gridControl1[1,1];
cell.Style.Value = "Hello";
```

Visual Basic:

```
Dim cell As Cell
cell = GridControl1(1, 1)
cell.Style.Value = "Hello"
```

The ***Cell*** class is included in the `Stingray.Grid` namespace. To reference the ***Cell*** class as shown above, this namespace must be opened for use as follows:

C#:

```
using Stingray.Grid;
```

Visual Basic:

```
Imports Stingray.Grid
```

Instances of the ***Cell*** class can be obtained only through an instance of the ***GridControl*** class. Once created, a cell instance maintains its association with the ***GridControl*** from which the cell was obtained. This allows the ***Cell*** instance to be used to change the properties of the referenced cell in the referenced grid. ***Cell*** instances can be used to change the style, text, formula, and other properties of a given grid cell. For example, the following code changes the cell at row 1 and column 2 so that it is read-only and is displayed using the Courier font.

C#:

```
cell = gridControl1[1,2];
Style style = gridControl1[1,2].Style;
style.ReadOnly = true;
style.TextFont = new OGFont("Courier", 10, 0);
style.Value = "World";
cell.Style = style;
```

Visual Basic:

```
cell = GridControl1(1, 2)
Dim style As Style
style = GridControl1(1, 2).Style
style.ReadOnly = true
style.TextFont = New OGFont("Courier", 10, 0)
style.Value = "World"
cell.Style = style
```

NOTE >> For efficiency, changes to individual attributes of the cell style are not applied to the grid cell immediately. Instead, style changes must be made as shown above. All desired changes to the cell style are performed in sequence. These changes are not visible in the grid cell until the cell style is applied back to the cell through an assignment of the updated cell style object.

For a complete description of the public interface to ***Cell***, see the *Objective Grid for Microsoft® .NET® Reference Guide*.

Public Methods

The ***Cell*** class contains the following public methods.

Table 4 – Cell Class Public Methods

Method	Description
`ApplyNewStyle`	Applies a new style stored in this cell object.
`ClearStyle`	Clears the embedded Style object.

Table 4 – Cell Class Public Methods (Continued)

Method	Description
`ClearValue`	Clears only the value.
`CopyCell`	Copies the contents of another cell into this cell.
`CopyCells`	Copies a range of cells.
`CopyStyle`	Copies the new Style over the one stored in this cell object.
`ExcludeStyle`	Attributes that are included in the source style are removed from the destination style.
`MoveCell`	Command for moving a cell. Cell references in formula expressions are adjusted if they depend on a moved cell.
`MoveCells`	Command for moving cells. Cell references in formula expressions are adjusted if they depend on cells in the moved range.
`OverrideStyle`	Override the current style in this cell object.
`ScrollInView`	Overloaded. Scrolls the grid so that this cell is in view.
`Select`	Overloaded. Selects or deselects this cell.

Public Properties

The ***Cell*** class contains the following public properties:

Table 5 – Cell Class Public Properties

Property	Description
`Col`	Gets the cell's column index.
`Formula`	Returns either the text value or formula for the cell.
`IsCurrent`	Determines if the cell is currently selected.
`Row`	Gets the cell's row index.
`Style`	Gets a reference to this cell's embedded Style object.

The Range Class

The ***Range*** class represents a rectangular range of cells in the grid. It can be set to represent an entire column, an entire row, or the entire table. Explicit coordinates can also be initialized with the top, bottom, left, and right members of a ***Range*** object.

Range objects appear frequently in the interface to ***GridControl***. ***Range***s are used for assigning styles and values to ranges of cells. ***Range***s are also used to hide, clear, move, copy, and select ranges of cells.

For a complete description of the public interface to ***Range***, see the *Objective Grid for Microsoft® .NET® Reference Guide*.

Public Methods

The ***Range*** class contains the following public methods.

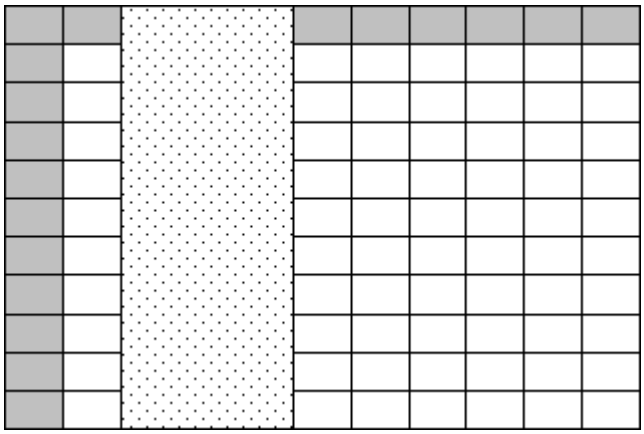
Table 6 – Range Class Public Methods

Method	Description
`Cell`	Creates a range referencing one cell
`Cells`	Creates an arbitrary range of cells
`Col`	Creates a range of one column
`Cols`	Creates a range of columns
`FromLTRB`	Creates a range from left, top, right, and bottom row and column indexes
`IntersectRange`	Determines if two ranges intersect
`InvalidRange`	Returns an empty range
`Row`	Returns a range consisting of one row
`Rows`	Returns a range of the specified rows
`Table`	Creates a range that references an entire table
`UnionRange`	Calculates the union of two ranges
`Clone`	Creates a copy of this Range
`Equals`	Determines if this range is equal to another range
`ExpandRange`	Sets the range based on the top, left corner of the range, and the number of rows and columns in the range.
`GetFirstCell`	Gets the first cell in a range
`GetHashCode`	Calculates the hash code for this range
`GetNextCell`	Overloaded. Gets the next cell in a range in row order
`InsertCols`	Inserts columns into a range at a specified starting location
`InsertRows`	Inserts rows into the range at the specified location
`IntersectRange`	Determines if this range intersects with a specified range
`IsCellInRange`	Returns `true` if and only if a specified cell coordinate is within this range

Use the `Cells()` method to specify a rectangular range of cells with well-defined top-left and bottom-right cells.

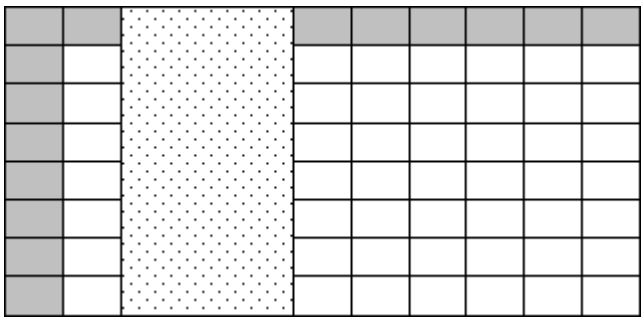
Use the `Rows()`, `Cols()`, and `Table()` methods to define regions without definite top-left or bottom-right coordinates. These methods define the range as being entire rows, entire columns, or the entire table, independent of the actual grid dimensions. This means that as the dimensions of the grid grow or shrink, the realized dimensions of the range also grow or shrink. For example, a grid with 10 rows, 10 columns, and a range defined using `Cols(2,4)` has the realized coordinates of top = 0, bottom = 10, left = 2, and right = 4, as shown in Figure 2.

Figure 2 – Range.Cols(2,4) in a 10x10 grid



If the last three rows of the grid are removed, the realized coordinates of the same range are now top = 0, bottom =7, left = 2, and right = 4. No changes were made to the original range. It is still defined as `Cols(2,4)`.

Figure 3 – Range.Cols(2,4) in a 7x10 grid



Ranges do not change when rows or columns are inserted into or deleted from the grid. In the example above, if we insert a new column between columns three and four, the range does not expand to reflect the addition. The range is still `Cols(2,4)`.

Calls to `Rows()`, `Cols()`, `Cells()`, and `Table()` are not cumulative. You cannot combine successive calls to these methods to define more complex range of cells. A call to any of these methods redefines a range. For example, calling `Rows(4,7)` followed by `Cols(2,5)` results in a range of `Cols(2,5)`, not the intersection or union of the two ranges.

Public Properties

The ***Range*** class contains the following public properties.

Table 7 – Range Class Public Properties

Property	Description
`Bottom`	Gets the bottom row in the range
`Height`	Gets the height, in rows, of the range
`IsCells`	Returns `true` if this is an arbitrary range of cells
`IsCols`	Returns `true` if this is a range of columns
`IsRows`	Returns `true` if this is a range of rows
`IsTable`	Returns `true` if this range references an entire table
`IsValid`	Returns `true` if and only if this range is valid
`Left`	Gets the leftmost column in the range
`RangeType`	Gets the type of range as a RangeType enumeration
`Right`	Gets the rightmost column in the range
`Top`	Gets the topmost row in the range
`Width`	Gets the width, in columns, of the range

The Style Class

The ***Style*** class encapsulates all of the information required to format a cell in a grid control. In addition to formatting information, styles also define most of the grid's behavior. The style information for a cell or a range of cells can be obtained from a ***Cell*** object. ***Style*** objects can also be applied to a cell or range of cells to change the format of that cell or range. The following code snippets show the use of a style object to format a range of cells in a grid control.

C#:

```
// Create a range and a style
Range range = Range.Cells(2, 3, 4, 5);
Style style = new Style();

// Assign a background color and value to the style
style.Interior = OGBrush.OGSolidBrush(Color.White);
style.Value = "Test";

// Apply the style to the grid
gridControl1.SetStyleRange(range, style);
```

Visual Basic:

```
' Create a range and a style
```

```
Dim range As Range
range = range.Cells(2, 3, 4, 5)
Dim style As Style
style = New Style()

' Assign a background color and value to the style
style.Interior = OGBrush.OGSolidBrush(Color.White)
style.Value = "Test"

' Apply the style to the grid
GridControl1.SetStyleRange(range, style)
```

The ***Style*** class supports combining style objects. For example, you can copy from one style to a second style only the properties that are not initialized in the second style. This feature allows you to combine styles and is analogous to the concept of inheritance in programming languages. You can specify a base style from which other styles can inherit properties at run time.

Properties of the ***Style*** class are associated with an include bit. This include bit is `true` when a property is initialized. If it is `false`, the property is not initialized. When drawing the grid, Objective Grid for Microsoft .NET fills up all uninitialized properties of the cell style object with values inherited from the base styles.

The Style attribute on a Cell object allows users to get and change the style of a specific cell. For details, see "The Cell Class." The `GridControl` class contains methods and events for changing the style of a cell or range of cells. For details, see "The GridControl Class."

For a complete description of the public interface to ***Style***, see the *Objective Grid for Microsoft® .NET® Reference Guide*.

Public Methods

The ***Style*** class contains the following public methods:

Table 8 – Style Class Public Methods

Method	Description
`Clone`	Creates a new copy of an existing Style object.
`Dispose`	Disposes of the resources used by a Style object.
`Free`	Frees the resources used by a Style object.
`SetDefaults`	Sets the cell's properties to default values.

Public Properties

The ***Style*** class contains the following public properties:

Table 9 – Style Class Public Properties

Property	Description
`AllowEnter`	Gets or sets the allow enter property.
`AutoSize`	Gets or sets the auto size property.
`Borders`	Gets or sets the appearance of the cell borders.
`ChoiceList`	Gets or sets choice list values.
`Control`	Gets or sets the embedded control type.
`Draw3dFrame`	Gets or sets the 3D effect of the cell.
`EllipseType`	Gets or sets the type of ellipsis used when the text is longer than the cell.
`Enabled`	Gets or sets the enabled state for the cell.
`FloatCell`	Gets or sets the float cell property.
`FloodCell`	Gets or sets the flood cell property.
`HorzAlign`	Gets or sets the cell's horizontal alignment.
`IncludeAllowEnter`	Gets or sets the state of the AllowEnter property.
`IncludeAutoSize`	Gets or sets the state of the AutoSize property.
`IncludeBorders`	Gets or sets the state of the Borders property.
`IncludeChoiceList`	Gets or sets the state of the ChoiceList property.
`IncludeControl`	Gets or sets the state of the Control property.
`IncludeDraw3dFrame`	Gets or sets the state of the Draw3dFrame property.
`IncludeEnabled`	Gets or sets the state of the Enabled property.
`IncludeFloatCell`	Gets or sets the state of the FloatCell property.
`IncludeFloodCell`	Gets or sets the state of the FloodCell property.
`IncludeHorzAlign`	Gets or sets the state of the HorzAlign property.
`IncludeInterior`	Gets or sets the state of the Interior property.
`IncludeItemData`	Gets or sets the state of the ItemData property.
`IncludeMaxLength`	Gets or sets the state of the MaxLength property.
`IncludeMergeCell`	Gets or sets the state of the MergeCell property.
`IncludeReadOnly`	Gets or sets the state of the ReadOnly property.

Table 9 – Style Class Public Properties (Continued)

Property	Description
`IncludeTextColor`	Gets or sets the state of the TextColor property.
`IncludeTextFont`	Gets or sets the state of the TextFont property.
`IncludeTriState`	Gets or sets the state of the TriState property.
`IncludeUserAttribute`	Gets or sets the state of a specific user attribute.
`IncludeValue`	Gets or sets the state of the Value property.
`IncludeVertAlign`	Gets or sets the state of the VertAlign property.
`IncludeVerticalScroll`	Gets or sets the state of the VerticalScroll property.
`IncludeWrapText`	Gets or sets the state of the WrapText property.
`InitialValue`	Gets or sets the initial value of the spin control.
`Interior`	Gets or sets the interior brush.
`ItemData`	Gets or sets a pointer to a user defined item data.
`LowerBound`	Gets or sets the lower bound of the spin control.
`Mask`	Gets or sets a user specified mask string.
`MaxLength`	Gets or sets the maximum length of the stored value.
`MergeCell`	Gets or sets the cell merge behavior.
`OnlyNumericValues`	Gets or sets numeric validation.
`Prompt`	Gets or sets the user specified input prompt.
`ReadOnly`	Gets or sets the cell's read-only state.
`TextColor`	Gets or sets the color of text in a cell.
`TextFont`	Gets or sets the cell's text font.
`ToolTip`	Gets or sets the string used for the tool tip.
`TriState`	Gets or sets the tristate property of the cell.
`TypeOf`	Gets or sets the value type.
`UpperBound`	Gets or sets the upper bound of the spin control.
`ValidateMessage`	Gets or sets the message to be displayed if the Style value is invalid.
`ValidMaximum`	Gets or sets the highest valid minimum value.
`ValidMinimum`	Gets or sets the lowest valid value.

Table 9 – Style Class Public Properties (Continued)

Property	Description
`Value`	Gets or sets the stored value.
`VertAlign`	Gets or sets the cell's vertical alignment.
`VerticalScroll`	Gets or sets the vertical scroll bar in a cell.
`WrapText`	Gets or sets text wrap.
`WrapValue`	Gets or sets the Wrap Value property.

Using the Style Class Efficiently

Making a change to a cell's style is an expensive operation. Because changes to a cell's style are automatically updated when assigned, making many changes sequentially can decrease your application's performance. The following code shows this penalty:

```
gridControl[1,1].Style.Value = "100";
gridControl[1,1].Style.TypeOf = StyleValueType.typeNumeric;
// ...other assignments to the Style object...
```

To avoid this penalty, create a new ***Style*** object (or copy an existing ***Style*** object), apply your changes to it, and then assign it to the cell(s):

```
// create a new Style object
Style myStyle = new Style();
// apply changes
myStyle.Value = "100";
myStyle.TypeOf = StyleValueType.typeNumeric;
// ...other assignments to myStyle ...
// assign new Style to a cell
gridControl[1,1].Style = myStyle;
```

This technique is particularly important when you are assigning many properties to the same cell.

The Param Class

An instance of the ***Param*** class contains or points to all necessary data for persisting the state of the grid object. Parameter objects can be stored in documents and serialized, or can be used as members of other classes. Parameter objects can also be shared between grid objects.

For a complete description of the public interface to ***Param***, see the *Objective Grid for Microsoft® .NET® Reference Guide*.

Table 10 – Public Instance Methods

Method	Description
`EnableTrackingColWidth`	Overloaded. Specifies the options for or disables changing column widths for the end user.
`EnableTrackingRowHeight`	Overloaded. Specifies the options for or disables changing row heights for the end user.
`Finalize`	Destructor.
`ReInitializeData`	Reinitializes the embedded OGData and OGParam objects.

Table 11 – Public Instance Properties

Property	Description
`CellActivationMode`	Determines the options for activating/editing the current cell.
`DrawOrder`	Gets or sets the drawing order.
`EnableHorizontalSorting`	Gets or sets row sorting when double clicked.
`EnableVerticalSorting`	Gets or sets column sorting when double clicked.
`ExcelLikeCurrentCell`	Gets or sets the default Objective Grid or Excel-like behavior of the current cell.
`ExcelLikeHeaders`	Gets or sets the Excel-like behavior of the headers.
`ExcelLikeScrolling`	Gets or sets scroll behavior to mimic Excel scroll behavior.
`ExcelLikeSelectionFrame`	Gets or sets the option to draw a small frame around the selected range of cells.
`ExpressionDisplayMode`	Gets or sets the display of formula expressions in inactive cells.
`HideCurrentCellMode`	Gets or sets options for hiding the current cell.
`LockReadOnly`	Gets or sets the readonly state for cells in the grid.
`NumberedColHeaders`	Gets or sets numbered column headers.

Table 11 – Public Instance Properties (Continued)

Property	Description
`NumberedRowHeaders`	Gets or sets numbered row headers.
`Properties`	Gets the OGProperties object associated with the grid.
`Selections`	Gets the allowable cell selections for the grid.
`SmartResize`	Enable and disable smart resizing.
`SpecialMode`	Gets or sets list box behavior.
`TransparentBackground`	Gets or sets a transparent background color for cells.

The GridControl Class

The ***GridControl*** class encapsulates the Objective Grid for Microsoft .NET custom control. It is derived from the ***Windows.Forms.Panel*** class (which is derived from ***System.Windows.Control***). It inherits a rich set of methods, properties, and events from its base class. It also adds a rich public interface, a number of properties, and many events of its own.

Windows Forms custom controls allow you to quickly create user interfaces by re-using existing controls. The Objective Grid for Microsoft .NET custom control is an example of the benefits of this programming paradigm. To use the ***GridControl***, you drag a ***GridControl*** from the Toolbox in Visual Studio onto a Windows form, and an instance of ***GridControl*** is created and displayed for design-time editing. The public properties and events defined for ***GridControl*** appear in the property grid for design-time manipulation.

For a complete description of the public interface to ***GridControl***, see the *Objective Grid for Microsoft® .NET® Reference Guide*.

Code Generated by the Windows Forms Designer

The Windows Forms Designer generates the code needed to instantiate, initialize, format (according to the selected design-time property settings), and display the grid. All of the following code is generated by the Windows Forms Designer:

C#:

```
#region Windows Forms Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
  this.gridControl1 = new Stingray.Grid.GridControl();
  ((System.ComponentModel.ISupportInitialize)(this.gridControl1)).
```

```
      BeginInit();
    this.SuspendLayout();
    //
    // gridControl1
    //
    this.gridControl1.ColCount = 30;
    this.gridControl1.DefaultColWidth = 70;
    this.gridControl1.DefaultRowHeight = 14;
    this.gridControl1.EnableUndo = true;
    this.gridControl1.FrozenCols = 0;
    this.gridControl1.FrozenRows = 0;
    this.gridControl1.HeaderCols = 0;
    this.gridControl1.HeaderRows = 0;
    this.gridControl1.Location = new System.Drawing.Point(8, 8);
    this.gridControl1.Name = "gridControl1";
    this.gridControl1.RowCount = 30;
    this.gridControl1.Size = new System.Drawing.Size(608, 472);
    this.gridControl1.TabIndex = 0;
    this.gridControl1.Text = "gridControl1";
    //
    // Form1
    //
    this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
    this.AutoScroll = true;
    this.ClientSize = new System.Drawing.Size(544, 405);
    this.Controls.AddRange(new System.Windows.Forms.Control[] {
      this.gridControl1});
    this.Name = "Form1";
    this.Text = "Sample Grid Application";
    ((System.ComponentModel.ISupportInitialize)(this.gridControl1)).
      EndInit();
    this.ResumeLayout(false);
  }
  #endregion
```

Visual Basic:

```
  'NOTE: The following procedure is required by the Windows Forms Designer
  'It can be modified using the Windows Forms Designer.
  'Do not modify it using the code editor.
  Friend WithEvents GridControl1 As Stingray.Grid.GridControl
  <System.Diagnostics.DebuggerStepThrough()>
  Private Sub InitializeComponent()
      Me.GridControl1 = New Stingray.Grid.GridControl()
      CType(Me.GridControl1,
        System.ComponentModel.ISupportInitialize).BeginInit()
      Me.SuspendLayout()
      '
      'GridControl1
      '
      Me.GridControl1.ColCount = 30
```

```
    Me.GridControl1.DefaultColWidth = 70
    Me.GridControl1.DefaultRowHeight = 15
    Me.GridControl1.EnableUndo = true
    Me.GridControl1.FrozenCols = 0
    Me.GridControl1.FrozenRows = 0
    Me.GridControl1.HeaderCols = 0
    Me.GridControl1.HeaderRows = 0
    Me.GridControl1.Location = New System.Drawing.Point(8, 8)
    Me.GridControl1.Name = "GridControl1"
    Me.GridControl1.RowCount = 30
    Me.GridControl1.Size = New System.Drawing.Size(584, 464)
    Me.GridControl1.TabIndex = 0
    Me.GridControl1.Text = "GridControl1"
    '
    'Form1
    '
    Me.AutoScaleBaseSize = New System.Drawing.Size(5, 13)
    Me.ClientSize = New System.Drawing.Size(600, 477)
    Me.Controls.AddRange(New System.Windows.Forms.Control()
      {Me.GridControl1})
    Me.Name = "Form1"
    Me.Text = "Form1"
    CType(Me.GridControl1,
      System.ComponentModel.ISupportInitialize).EndInit()
    Me.ResumeLayout(false)
End Sub
```

Events

GridControl defines a number of public events along with associated virtual event handlers. This allows you to handle grid events in one of two ways.

1. You can derive a new control class from ***GridControl***, override the virtual event handler methods, and include custom event-handling logic for the events. This technique must be used when you want to completely avoid executing the base class (***GridControl***) event-handling logic for an event. Do not call the base class event handler.

   NOTE >> The base class event handler must be called in order to invoke any delegates attached to the event.

2. You can attach delegates to the events and handle the events in those delegates. This mechanism is convenient, particularly when using the Windows Forms Designer. You can double-click the name of the event to be handled, and the Windows Forms Designer automatically generates a delegate stub and attaches it to the event. You then write the event-handling logic in the generated delegate stub.

NOTE >> Using delegates does not give you any control over how or when the base class event-handling logic is executed (except in special cases where event-handling options are included in the event arguments for the event).

Once the ***GridControl*** is placed into a Windows Form and all desired design-time configurations are complete, you can use the public interface to the ***GridControl*** to programmatically control the grid and respond to grid events.

Public Methods

This section describes the methods of the ***GridControl*** class. Table 12 lists public and protected methods available for ***GridControl***.

Table 12 – GridControl Public and Protected Methods

Method	Description
`BeginInit`	Implements ISupportInitialize.
`CalcClientColFromPt`	Determines the column for the point.
`CalcClientRowFromPt`	Determines the row for the point.
`CalcRectFromRowColEx`	Overloaded. Computes the window-area for the given range of cells.
`CalcSumOfColWidths`	Overloaded. Computes the total width of the given columns. If the total width exceeds nAbortAt, the method aborts and returns the value that is greater than nAbortAt.
`CalcSumOfRowHeights`	Overloaded. Computes the total height of the given rows.
`CanClear`	Determines if the currently selected range of cells can be cleared.
`CanCopy`	Determines if the currently selected range of cells can be copied.
`CanCut`	Determines if the currently selected range of cells can be cut.
`CanPaste`	Determines if a range of cells on the clipboard can be pasted into the grid.
`CanSelectCurrentCell`	Determines if the new current cell can be selected or the old current cell deselected.
`ClearCells`	Overloaded. Clears a range of cells.

Table 12 – GridControl Public and Protected Methods (Continued)

Method	Description
`ConvertClientCol`	Converts the specified column index into a relative index.
`ConvertClientRow`	Converts the specified row index into a relative index.
`ConvertCol`	Converts the specified column index into an absolute index.
`ConvertRow`	Converts the specified row index into an absolute index.
`CopyCells`	Overloaded. Copies a range of cells to a specified location in the grid.
`DelayFloatCells`	Overloaded. Recalculates floating cells.
`DelayMergeCells`	Overloaded. Recalculates merge cells.
`DesignTimeInitialize`	Performs some specialized design-time grid initialization.
`EndInit`	Implements ISupportInitialize.
`GetColWidth`	Gets the column width, in pixels, of the specified column. If it is 0 then the column is hidden.
`GetExpressionRowCol`	Gets the expression stored in a particular cell.
`GetFontHeight`	Returns the height of the standard-style font in pixels.
`GetFontWidth`	Returns the average width of the standard-style font in pixels.
`GetGridRect`	Returns the window area for the grid.
`GetRowHeight`	Gets the row height, in pixels, of the specified column. If it is 0, the row is hidden.
`GetSelectedCols`	Overloaded. Returns an array containing the selected column IDs.
`GetSelectedRows`	Overloaded. Returns an array containing the selected row IDs.
`GetValueRowCol`	Gets the value stored in a cell.
`Height_DPtoLP`	Divides the pixel-value by the value determined through GetFontHeight.
`Height_LPtoDP`	Multiplies the logical value with the value determined through GetFontHeight.

Table 12 – GridControl Public and Protected Methods (Continued)

Method	Description
`HideCols`	Overloaded. Hides the specified columns.
`HideRange`	Hides a range of cells.
`HideRows`	Overloaded. Hides the specified rows.
`HitTest`	Determines the type of information displayed at a window coordinate (pt).
`Initialize`	Initializes a GridControl object state at runtime.
`InitParamObject`	Initializes the grid parameter object.
`InsertCols`	Overloaded. Inserts columns.
`InsertRows`	Overloaded. Inserts rows.
`IsActiveCurrentCell`	Returns `true` if the current cell is active.
`IsColHidden`	Determines if the specified column is hidden.
`IsCurrentCell`	Overloaded. Determines if a current cell is selected or is the specified cell.
`IsRowHidden`	Determines if the specified row is hidden.
`LockUpdate`	Overloaded. Prevents display of updates until `lockUpdate(false)` is called.
`MoveCells`	Overloaded. Moves a range of cells.
`MoveCols`	Overloaded. Moves the specified block of columns to another location.
`MoveCurrentCell`	Overloaded. Selects a new current cell based on an offset from the existing current cell.
`MoveRows`	Overloaded. Moves the specified block of rows to another location.
`OnClear`	Overloaded. Raises the Clear event.
`OnCopy`	Raises the Copy event.
`OnCut`	Raises the Cut event.
`OnFind`	Raises the Find event.
`OnGetStyleRowCol`	Raises the GetStyleRowCol event.
`OnPaste`	Raises the Paste event.
`OnPrint`	Raises the Print event.
`OnPrintPreview`	Overloaded. Raises the PrintPreview event.
`OnRedo`	Raises the Redo event.

Table 12 – GridControl Public and Protected Methods (Continued)

Method	Description
`OnReplace`	Raises the Replace event.
`OnUndo`	Raises the Undo event.
`RedrawGrid`	Overloaded. Redraws the whole grid.
`RedrawRowCol`	Overloaded. Redraw the specified range of cells.
`RegisterControl`	Registers a .NET control for use as a Objective Grid for Microsoft .NET cell editor.
`RemoveCols`	Overloaded. Removes a block of columns.
`RemoveRows`	Overloaded. Removes a block of rows.
`ResetCurrentCell`	Overloaded. Deactivates the current cell.
`ScrollCellInView`	Overloaded. Scrolls the cell into the view if it is outside the visible area.
`SelectRange`	Overloaded. Selects or deselects a range of cells.
`SetColCount`	nColsthe number of columns for the grid rFlags-the RedrawFlags for the operation
`SetColWidth`	Overloaded. Sets the column-widths for specific columns in pixels.
`SetCoveredCellsRowCol`	Overloaded. Sets the covered cells-range for a cell.
`SetExpressionRowCol`	Overloaded. Stores the expression in the specified cell.
`SetFloatCellsMode`	Turns on and off the calculation of floating cells in the grid.
`SetGridLineStyle`	Set the grid line style.
`SetGridRect`	Overloaded. Sets the grid rectangle for the window.
`SetMergeCellsMode`	Turns on and off the calculation of merge cells in the grid.
`SetRowCount`	Sets the number of rows in the grid.
`SetRowHeight`	Overloaded. Sets the row-heights for specific rows in pixels.
`SetStyleRange`	Overloaded. Applies cell formatting to the specified range of cells.

Table 12 – GridControl Public and Protected Methods (Continued)

Method	Description
`SetValueRange`	Overloaded. Sets a range of cells with the specified value.
`TransferCurrentCell`	Overloaded. Stores and deactivates the current cell or actualizes the current cell's contents.
`UseControl`	Uses the named CellControl in the grid at the named row and column location.
`Width_DPtoLP`	The method divides the pixel-value by the value determined through GetFontWidth.
`Width_LPtoDP`	The method multiplies the logical value with the value determined through GetFontWidth.
`Finalize`	Destructor.
`UpdateFrozenCols`	Overloaded. Updates the window after freezing columns.
`UpdateFrozenRows`	Overloaded. Updates the window after freezing rows.

Table 13 lists the virtual event handlers that are available with ***GridControl***. Each of these methods raises its corresponding event (to invoke any attached delegates) when it is invoked. When overriding these methods, the base (***GridControl***) implementation of the method must be invoked from the derived class implementation, in order to invoke any attached delegates.

Table 13 – GridControl Virtual Event Handlers

Event Handler	Description
`OnCanceledEditing`	Raises the CanceledEditing event.
`OnCancelEditing`	Raises the CancelEditing event.
`OnClickedButtonRowCol`	Raises the ClickedButtonRowCol event.
`OnCreateControl`	Overridden from Panel.
`OnDeleteCell`	Raises the DeleteCell event.
`OnDrawGridItem`	Raises the DrawGridItem event.
`OnDrawTopLeftBottomRight`	Raises the DrawTopLeftBottomRight event.
`OnEndEditing`	Raises the EndEditing event.
`OnGridDraw`	Raises the GridDraw event.
`OnInitCurrentCell`	Raises the InitCurrentCell event.

Table 13 – GridControl Virtual Event Handlers (Continued)

Event Handler	Description
`OnLButtonClickedRowCol`	Raises the LButtonClickedRowCol event.
`OnLButtonDblClkRowCol`	Raises the LButtonDblClkRowCol event.
`OnLButtonHitRowCol`	Raises the LButtonHitRowCol event.
`OnLeftCell`	Raises the LeftCell event.
`OnMButtonClickedRowCol`	Raises the MButtonClickedRowCol event.
`OnMButtonDblClkRowCol`	Raises the MButtonDblClkRowCol event.
`OnMButtonHitRowCol`	Raises the MButtonHitRowCol event.
`OnModifyCell`	Raises the ModifyCell event.
`OnMovedCurrentCell`	Raises the MovedCurrentCell event.
`OnProcessKeys`	Raises the ProcessKeys event.
`OnRButtonClickedRowCol`	Raises the RButtonClickedRowCol event.
`OnRButtonDblClkRowCol`	Raises the RButtonDblClkRowCol event.
`OnRButtonHitRowCol`	Raises the RButtonHitRowCol event.
`OnStartEditing`	Raises the StartEditing event.
`OnStoreColWidth`	Raises the StoreColWidth event.
`OnStoreCopyCells`	Raises the StoreCopyCells event.
`OnStoreDefaultColWidth`	Raises the StoreDefaultColWidth event.
`OnStoreDefaultRowHeight`	Raises the StoreDefaultRowHeight event.
`OnStoreFrozenCols`	Raises the StoreFrozenCols event.
`OnStoreFrozenRows`	Raises the StoreFrozenRows event.
`OnStoreHideCol`	Raises the StoreHideCol event.
`OnStoreHideRow`	Raises the StoreHideRow event.
`OnStoreInsertCols`	Raises the StoreInsertCols event.
`OnStoreInsertRows`	Raises the StoreInsertRows event.
`OnStoreMoveCols`	Raises the StoreMoveCols event.
`OnStoreMoveRows`	Raises the StoreMoveRows event.
`OnStoreReadOnly`	Raises the StoreReadOnly event.
`OnStoreRemoveCols`	Raises the StoreRemoveCols event.
`OnStoreRemoveRows`	Raises the StoreRemoveRows event.
`OnStoreRowHeight`	Raises the StoreRowHeight event.

Table 13 – GridControl Virtual Event Handlers (Continued)

Event Handler	Description
`OnStoreStyleRowCol`	Raises the StoreStyleRowCol event.
`OnStoreZoom`	Raises the StoreZoom event.
`OnValidateCell`	Raises the ValidateCell event.
`OnValidating`	Overridden from Panel.

Public Properties

Table 14 shows the public properties in the ***GridControl*** class.

Table 14 – GridControl Public Properties

Property	Description
`AcclerateArrowKey`	Sets scrolling speed with arrow keys.
`AcclerateScrollBars`	Sets scrolling speed with scrollbar buttons.
`AutoScroll`	Sets auto-scroll mode.
`Cell`	Grid cell indexer, which allows a grid Cell object to be obtained.
`CurrentCell`	Gets or sets the current grid cell.
`DefaultColWidth`	Returns the default value for the column-width in pixels.
`DefaultRowHeight`	Returns the default value for the row-height in pixels.
`EnableUndo`	Returns `true` if undo-creation is enabled; `false` otherwise.
`Features`	Returns a reference to the GridFeatures object associated with a particular GridControl object.
`FloatCells`	Returns the setting of floating cells: enabled or disabled.
`FrozenCols`	Returns the number of frozen columns in the grid.
`FrozenRows`	Returns the number of frozen rows.
`HeaderCols`	Returns the number of columns to be used as row headers.
`HeaderRows`	Returns the number of rows to be used as column headers.

Table 14 – GridControl Public Properties (Continued)

Property	Description
`LeftCol`	Returns the leftmost non-frozen column of the view.
`MergeCells`	Returns the setting of merging cells: enabled or disabled.
`Param`	Returns a pointer to the parameter object.
`Properties`	Returns a pointer to the properties object.
`ReadOnly`	Returns the read only setting.
`RightToLeft`	Overrides Control.RightToLeft.
`TopRow`	Returns the index of the topmost scrollable row.
`Zoom`	Returns the zooming-factor as a percentage.

Public Events

This section lists the events available to be handled in the ***GridControl*** class.

Each event has an associated virtual event handler, whose name is prefixed with "On". The grid-specific event handlers (those not inherited from the ***Windows.Forms.Control*** class) accept an event arguments object whose type name is formed by the name of the event plus the text "EventArgs". For example, the CancelEditing event has an associated event handler named "OnCancelEditing", and the argument passed to "OnCancelEditing" is of type "CancelEditingEventArgs".

A delegate attached to grid-specific events accepts two arguments: an object and an event arguments object of the same type as supplied to the virtual event handler for the event. For example, the signature of a delegate for the CancelEditing event would appear as follows:

C#:

```
private void gridControl1_CancelEditing(
             object sender,
             Stingray.Grid.CancelEditingEventArgs e)
{ }
```

Visual Basic:

```
Private Sub GridControl1_CancelEditing(
            ByVal sender As Object,
            ByVal e As Stingray.Grid.CancelEditingEventArgs)
            Handles GridControl1.CancelEditing

End Sub
```

For more information regarding the events and virtual event handlers associated with the ***GridControl*** class, see the *Objective Grid for Microsoft® .NET® Reference Guide*.

Table 15 lists the Objective Grid for Microsoft .NET-specific public events. The events are grouped by function.

Table 15 – GridControl Public Events

Event	Description
`CanceledEditing`	Canceled Editing Event.
`CancelEditing`	Cancel Editing Event.
`Clear`	Clear Event.
`Click`	Occurs when the control is clicked.
`ClickedButtonRowCol`	Clicked Button Row/Column Event.
`Copy`	Copy Event.
`Cut`	Cut Event.
`DeleteCell`	Delete Cell Event.
`DrawGridItem`	Draw Grid Item Event.
`DrawTopLeftBottomRight`	Draw Top, Left, Bottom, Right Event.
`EndEditing`	End Editing Event.
`Find`	Find Event.
`GetStyleRowCol`	Get Style Row Col Event.
`GridBeginPrint`	Invoked when printing begins.
`GridBeginPrintPreview`	Invoked when print preview begins.
`GridDraw`	Grid Draw Event.
`GridEndPrint`	Invoked when printing is ending.
`GridEndPrintPreview`	Invoked when print preview is ending.
`GridInitialized`	Grid Initialized Event.
`GridPrintPage`	Invoked as each page is printed.
`GridPrintPreviewPage`	Invoked as each page of a print preview is prepared for preview.
`HorizontalScroll`	Horizontal Scroll Event.
`InitCurrentCell`	Init Current Cell Event.
`KeyDown`	Occurs when a key is pressed while the control has focus.
`KeyPress`	Occurs when a key is pressed while the control has focus.
`KeyUp`	Occurs when a key is released while the control has focus.

Table 15 – GridControl Public Events (Continued)

Event	Description
`LButtonClickedRowCol`	Left Button Clicked Row Column Event.
`LButtonDblClkRowCol`	Left Button Double Clicked Row Column Event.
`LButtonHitRowCol`	Left Button Hit Row Column Event.
`LeftCell`	Left Cell Event.
`MButtonClickedRowCol`	Middle Button Clicked Row Column Event.
`MButtonDblClkRowCol`	Middle Button Double Clicked Row Column Event.
`MButtonHitRowCol`	Middle Button Hit Row Column Event.
`ModifyCell`	Modify Cell Event.
`MouseDown`	Occurs when the mouse pointer is over the control and a mouse button is pressed.
`MouseMove`	Occurs when the mouse pointer is moved over the control.
`MouseUp`	Occurs when the mouse pointer is over the control and a mouse button is released.
`MovedCurrentCell`	Moved Current Cell Event.
`Paint`	Occurs when the control is redrawn.
`Paste`	Paste Event.
`Print`	Print event invoked from OnPrint.
`PrintPreview`	PrintPreview event invoked from OnPrintPreview.
`ProcessKeys`	Process Keys Event.
`RButtonClickedRowCol`	Right Button Clicked Row Column Event.
`RButtonDblClkRowCol`	Right Button Double Clicked Row Column Event.
`RButtonHitRowCol`	Right Button Hit Row Column Event.
`Redo`	Redo Event.
`Replace`	Replace Event.
`StartEditing`	Start Editing Event.

Table 15 – GridControl Public Events (Continued)

Event	Description
`StoreColWidth`	Store Column Width Event.
`StoreCopyCells`	Store Copy Cells Event.
`StoreDefaultColWidth`	Store Default Column Width Event.
`StoreDefaultRowHeight`	Store Default Row Height Event.
`StoreFrozenCols`	Store Frozen Cells Event.
`StoreFrozenRows`	Store Frozen Rows Event.
`StoreHideCol`	Store Hide Column Event.
`StoreHideRow`	Store Hide Row Event.
`StoreInsertCols`	Store Insert Columns Event.
`StoreInsertRows`	Store Insert Rows Event.
`StoreMoveCols`	Store Move Columns Event.
`StoreMoveRows`	Store Move Rows Event.
`StoreReadOnly`	Store ReadOnly Event.
`StoreRemoveCols`	Store Remove Columns Event.
`StoreRemoveRows`	Store Remove Rows Event.
`StoreRowHeight`	Store Row Height Event.
`StoreStyleRowCol`	Store Style Row Column Event.
`StoreZoom`	Store Zoom Event.
`Undo`	Undo Event.
`ValidateCell`	Validate Cell Event.
`VerticalScroll`	Vertical Scroll Event.

Predefined Controls

Objective Grid for Microsoft .NET includes a number of pre-defined controls that can be inserted into grid cells. These controls are used in a cell by setting a property in the cell's style. The `ControlType` enumeration contains identifiers for all of the pre-defined controls available for use, and the `Control` property of the ***Style*** class inserts a custom control into a cell.

The following code snippets show how to insert a drop-down control into a grid cell. The drop-down control contains the items "Rogue", "Wave", and "Stingray".

C#:

```
Style style = new Style();
style.Control = Stingray.Grid.ControlType.DropDown;
style.ChoiceList = "Rogue\nWave\nStingray";
gridControl1[1,1].Style = style;
```

Visual Basic:

```
Dim style As New Style()
style.Control = Stingray.Grid.ControlType.DropDown
Dim str As New System.Text.StringBuilder()
str.Append("Rogue")
str.Append(Chr(10))
str.Append("Wave")
str.Append(Chr(10))
str.Append("Stingray")
str.Append(Chr(10))
style.ChoiceList = str.ToString()
GridControl1(1, 1).Style = style
```

Table 16 lists the custom controls available in Objective Grid for Microsoft .NET, and their corresponding `ControlType` enumerations.

Table 16 – Predefined Controls

Control	Description
`Label`	Static Label Control
`Header`	Header Control similar to Row and Column Headers
`TextBox`	TextBox Control
`SpinEdit`	Spin Control
`ScrollEdit`	TextBox with a Scroll Bar
`HotSpotEdit`	TextBox with a HotSpot Button
`RichEdit`	RichEdit Control, helps formatting individual characters

Table 16 – Predefined Controls (Continued)

Control	Description
`MaskEdit`	Standard Mask Edit Control
`Password`	Password Edit control
`Button`	Push button
`RadioButton`	Radio button
`Checkbox`	Check box control
`ListBox`	List box
`DropDown`	Drop down
`DropDownList`	Drop down list
`TabbedDropDown`	Tabbed drop down
`TabbedDropDownList`	Tabbed drop down list
`ComboBox`	ComboBox
`ValidatedComboBox`	ComboBox, permits entering a string, which is available as one of the choices
`TabbedComboBox`	Tabbed combo box
`CheckListComboBox`	ComboBox with CheckBox for each choice
`ZeroBasedComboBox`	Returns the index of the selected choice with zero as the starting index
`OneBasedComboBox`	Returns the index of the selected choice with one the starting index
`ZeroBasedComboBoxEx`	Zero-based combo box
`OneBasedComboBoxEx`	One-based combo box
`DateTime`	DateTime Control with Calendar for input
`DateTimeNoCalendar`	DateTime Control without Calendar for input
`Currency`	Currency Text Box
`ProgressBar`	Progress bar

Browser Architecture

Objective Grid for Microsoft .NET contains classes that allow a grid to be tied to an ADO.NET data source.

Formula Engine Support

Objective Grid for Microsoft .NET supports the use of formulas in a grid. Formula support is controlled through two properties on the ***GridControl***'s `Features` property. These two properties are `EnableFormulaEngine` and `EnableWorkSheetFunctions`. `EnableFormulaEngine` enables the formula engine. `EnableWorkSheetFunctions` enables all of the built-in worksheet functions supplied with Objective Grid for Microsoft .NET.

Enabling Formula Support

To enable formula support and built-in functions at design time:

1. Click the ***GridControl*** in your Windows Form.
2. In the property grid, expand the `Features` property in the "Configurations" category.
3. Set the `EnableFormulaEngine` and `EnableWorkSheetFunctions` properties to `true`.

The following code is generated by the Windows Forms Designer.

C#:

```
this.gridControl1.Features.EnableFormulaEngine = true;
this.gridControl1.Features.EnableWorkSheetFunctions = true;
```

Visual Basic:

```
Me.GridControl1.Features.EnableFormulaEngine = true
Me.GridControl1.Features.EnableWorkSheetFunctions = true
```

After formula support is enabled, formulas can be used in grid cells. For example, the following code computes the sum of the values in cells A1 through A3:

C#:

```
gridControl1[1,1].Style.Value = "1";
gridControl1[2,1].Style.Value = "2";
gridControl1[3,1].Style.Value = "3";
gridControl1[4,1].Formula = "=SUM(A1:A3)";
```

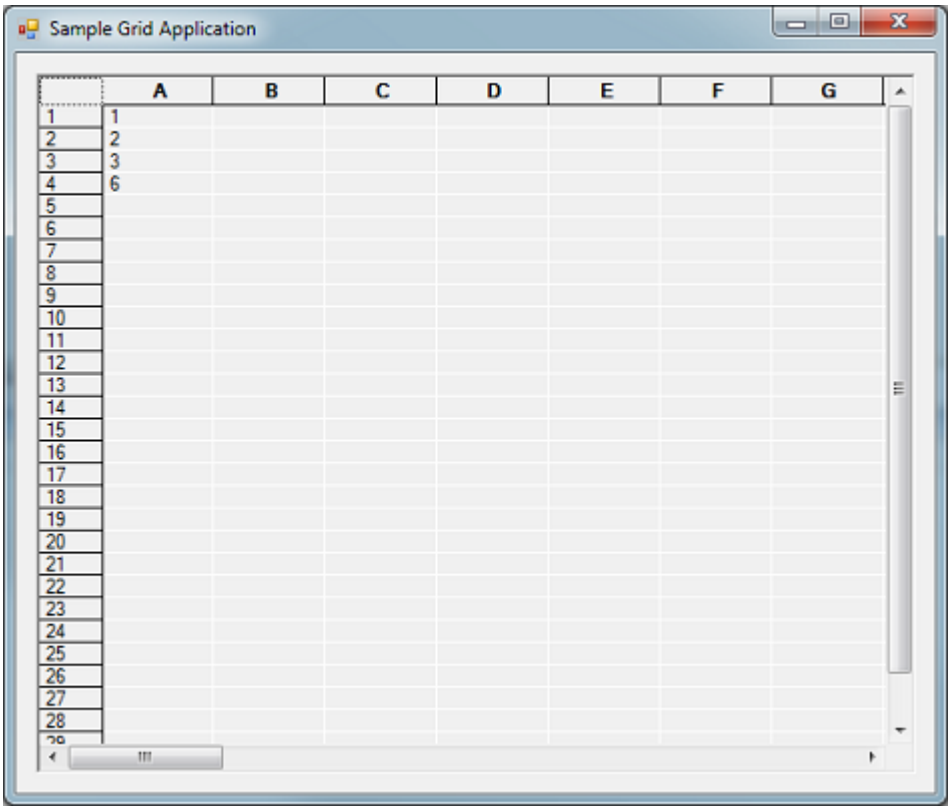
Visual Basic:

```
GridControl1(1, 1).Style.Value = "1"
GridControl1(2, 1).Style.Value = "2"
```

```
GridControl1(3, 1).Style.Value = "3"
GridControl1(4, 1).Formula = "=SUM(A1:A3)"
```

The result is shown in Figure 4, “Using the Formula Engine.”

Figure 4 – Using the Formula Engine



Mathematical Functions

Table 17 lists the mathematical functions that are supported.

Table 17 – Mathematical Functions

Function	Description
=ABS(X)	The absolute value of X.
=ACOS(X)	The arc cosine of X.
=ASIN(X)	The arc sine of X.
=ATAN(X)	The 2-quadrant arc tangent of X.
=ATAN2(X, Y)	The 4-quadrant arc tangent of Y/X.
=CEIL(X)	The smallest integer greater than or equal to X.

Table 17 – Mathematical Functions (Continued)

Function	Description
=COS(X)	The cosine of X.
=COSH(X)	The hyperbolic cosine of X.
=DEGREES(X)	Converts the angle expressed in radians to degrees ( ).
=DET(M)	The determinant of the matrix range M, which must be a square matrix.
=DOT(R1, R2)	The dot product of the vectors R1 and R2.
=EXP(X)	e raised to the X power.
=FACT(N)	The value of N!.
=FLOOR(X)	The largest integer less than or equal to X.
=FRAC(X)	The fractional portion of X.
=GAMMA(X)	The value of the gamma function evaluated at X.
=GRAND	A 12th-degree binomial approximation to a Gaussian random number with zero mean and unit variance.
=INT(X)	The integer portion of X.
=LN(X)	The natural log (base e) of X.
=LNGAMMA(X)	The log base e of the gamma function evaluated at X.
=LOG(X)	The log base 10 of X.
=LOG10(X)	The log base 10 of X.
=LOG2(X)	The log base 2 of X.
=MOD(X, Y)	The remainder of X/Y.
=MODULUS(X, Y)	The modulus of X/Y.
=PI	The value of pi.
=POLY(X, ...)	The value of an Nth-degree polynomial in X.
=PRODUCT(X, ...)	The product of all the numeric values in the argument list.
=RADIANS(X)	Converts the angle expressed in degrees to radians ( ).
=RAND	A uniform random number on the interval (0,1).
=ROUND(X, n)	X rounded to n number of decimal places (0 to 15).

Table 17 – Mathematical Functions (Continued)

Function	Description
=SIGMOID(X)	The value of the sigmoid function.
=SIN(X)	The sine of X.
=SINH(X)	The hyperbolic sine of X.
=SQRT(X)	The positive square root of X.
=SUMPRODUCT(R1, R2)	The dot product of the vectors R1 and R2, where R1 and R2 are of equal dimension.
=TAN(X)	The tangent of X.
=TANH(X)	The hyperbolic tangent of X.
=TRANSPOSE(M)	The transpose of matrix M.
=VECLEN(...)	The square root of the sum of squares of its arguments.

Statistical Functions

Table 18 lists statistical functions that are supported.

Table 18 – Statistical Functions

Function	Purpose
=AVG(...)	The average (arithmetic mean) of its arguments.
=CORR(R1, R2)	Pearson's product-moment correlation coefficient for the paired data in ranges R1 and R2.
=COUNT(...)	A count of its non-blank arguments.
=F(M, N, F)	The integral of Snedecor's F-distribution with M and N degrees of freedom from minus infinity to F.
=ERF(L[, U])	Error function integrated between 0 and L; if U specified, between L and U.
=ERFC(L)	Complementary error function integrated between L and infinity.
=FORECAST(...)	Predicted Y values for given X.
=FTEST(R1, R2)	The significance level ( ) of the two-sided F-test on the variances of the data specified by ranges R1 and R2.
=GMEAN(...)	The geometric mean of its arguments.
=HMEAN(...)	The harmonic mean of its arguments.

Table 18 – Statistical Functions (Continued)

Function	Purpose
=LARGE(R, N)	The Nth largest value in range R.
=MAX(...)	The maximum of its arguments.
=MEDIAN(...)	The median (middle value) of the range R1.
=MIN(...)	The minimum of its arguments.
=MODE(...)	The mode or most frequently occurring value.
=MSQ(...)	The mean of the squares of its arguments.
=PERCENTILE(R, N)	The value from the range R that is at the Nth percentile in R.
=PERCENTRANK(R, N)	The percentile rank of the number N among the values in range R.
=PERMUT(S, T)	The number of T objects that can be chosen from the set S, where order is significant.
=PTTEST(R1, R2)	The significance level ( ) of the two-sided T-test for the paired samples contained in ranges R1 and R2.
=QUARTILE(R, Q)	The quartile Q of the data in range R.
=RANK(E, R[, O])	The rank of a numeric argument E in the range R.
=SSQ(...)	The sum of squares of its arguments.
=RMS(...)	The root of the mean of squares of its arguments.
=SMALL(R, N)	The Nth smallest number in range R.
=SSE(...)	The sum squared error of its arguments. It is equivalent to =VAR(...) =COUNT(...).
=STD(...)	The population standard deviation (N weighting) of its arguments.
=STDS(...)	The sample standard deviation (N-1 weighting) of its arguments.
=SUM(...)	The sum of its arguments.
=T(N, T)	The integral of Student's T-distribution with N degrees of freedom from minus infinity to T.
=TTEST(R, X)	The significance level of the two-sided single population T-test for the population samples contained in range R.

Table 18 – Statistical Functions (Continued)

Function	Purpose
=TTEST2EV(R1, R2)	The significance level ( ) of the two-sided dual population T-test for ranges R1 and R2, where the population variances are equal.
=TTEST2UV(R1, R2)	The significance level ( ) of the two-sided dual population T-test for ranges R1 and R2, where the population variances are not equal.
=VAR(...)	The sample variance (N weighting) of its arguments.
=VARS(...)	The sample variance (N-1 weighting) of its arguments.
=VSUM(...)	The visual sum of its arguments, using precision and rounding of formatted cell values.

Conditional Statistical Functions

Table 19 lists conditional statistical functions that are supported.

Table 19 – Conditional Statistical Functions

Function	Purpose
=CAVG(..., C)	Conditional average.
=CCOUN(..., C)	Conditional count.
=CMAX(..., C)	Conditional maximum.
=CMIN(..., C)	Conditional minimum.
=CSTD(..., C)	Conditional sample standard deviation (N weighting).
=CSTDS(..., C)	Conditional sample standard deviation (N-1 weighting).
=CSUM(..., C)	Conditional sum.
=CVAR(..., C)	Conditional population variance (N weighting).
=CVARS(..., C)	Conditional population variance (N-1 weighting).

String Functions

Table 20 lists string functions that are supported.

Table 20 – String Functions

Function	Purpose
=CHAR(N)	The character represented by the code N.
=CLEAN(S)	The string formed by removing all non-printing characters from the string S.
=CODE(S)	The ASCII code for the first character in string S.
=EXACT(S1, S2)	Returns `true` (1) if string S1 exactly matches string S2, otherwise returns 0.
=FIND(S1, S2, N)	The index of the first occurrence of S1 in S2.
=HEXTONUM(S)	The numeric value for the hexadecimal interpretation of S.
=LEFT(S, N)	The string composed of the leftmost N characters of S.
=LENGTH(S)	The number of characters in S.
=LOWER(S)	S converted to lower case.
=MID(S, N1, N2)	The string of length N2 that starts at position N1 in S.
=NUMTOHEX(X)	The hexadecimal representation of the integer portion of X.
=PROPER(S)	The string S with the first letter of each word capitalized.
=REGEX(S1, S2)	Returns `true` (1) if string S1 exactly matches string S2; otherwise returns `false` (0). Allows "wildcard"' comparisons by interpreting S1 as a regular expression.
=REPEAT(S, N)	The string S repeated N times.
=REPLACE(S1, N1, N2, S2)	The string formed by replacing the N2 characters starting at position N1 in S1 with string S2.
=RIGHT(S, N)	The string composed of the rightmost N characters of S.
=STRCAT(...)	The concatenation of all its arguments.
=STRING(X, N)	The string representing the numeric value of X, to N decimal places.

Table 20 – String Functions (Continued)

Function	Purpose
=STRLEN(...)	The total length of all strings in its arguments.
=TRIM(S)	The string formed by removing spaces from the string S.
=UPPER(S)	The string S converted to upper case.
=VALUE(S)	The numeric value represented by the string S; otherwise 0 if S does not represent a number.

Logic Functions

Table 21 lists the supported logic functions.

Table 21 – Logic Functions

Function	Purpose
=FALSE	The logical value 0.
=FILEEXISTS(S)	1 if file S can be opened for reading; otherwise 0.
=IF(X, T, F)	The value of T if X evaluates to 1, or F if X evaluates to 0.
=ISERROR(X)	Returns 1 if X "contains" an error, otherwise 0.
=ISNUMBER(X)	1 if X is a numeric value; otherwise 0.
=ISSTRING(X)	1 if X is a string value; otherwise 0.
=TRUE	The logical value 1.
=AND(...)	0 if any arguments are 0; 1 if all arguments are 1; otherwise -1.
=NAND(...)	0 if all arguments are 1; 1 if any arguments are 0; otherwise -1.
=NOR(...)	0 if any arguments are 1; 1 if all arguments are 0; otherwise -1.
=NOT(X)	0 if X=1; 1 if X=0; otherwise -1.
=OR(...)	0 if all arguments are 0; 1 if any arguments are 1; otherwise -1.
=XOR(...)	-1 if any of the arguments are not 0 or 1; otherwise 0 if the total number of arguments with the value 1 is even; 1 if the total number of arguments with the value 1 is odd.

Financial Functions

Table 22 lists the supported financial functions.

Table 22 – Financial Functions

Function	Description
=COUPDAYBS(S, M, F[, B])	The number of days between the beginning of the coupon period to the settlement date.
=ACCRINT(I, Ft, S, R, P, F[, B])	Accrued interest for a security that pays periodic interest.
=ACCRINTM(I, S, R, P[, B])	Accrued interest for a security that pays interest at maturity.
=COUPDAYS(S, M, F[, B])	The number of days in the coupon period that the settlement date is in.
=COUPDAYSNC(S, M, F[, B])	The number of days between the settlement date and the next coupon date.
=COUPNCD(S, M, F[, B])	The next coupon date after the settlement date.
=COUPNUM(S, M, F[, B])	The number of coupon payments between the settlement date and maturity date.
=COUPPCD(S, M, F[, B])	The previous (most recent) coupon date before the settlement date.
=CTERM(R, FV, PV)	The number of compounding periods for an investment.
=CUMIPMT(R, NP, PV, S, E, T)	The cumulative interest on a loan between start period S and end period E.
=CUMPRINC(R, NP, PV, S, E, T)	The cumulative principal paid on a loan between start period S and end period E.
=DB(C, S, L, P[, M])	Fixed-declining depreciation allowance.
=DDB(C, S, L, N)	Double-declining depreciation allowance.
=DISC(S, M, P, R[, B])	The discount rate for a security.
=DOLLARDE(FD, F)	Converts a dollar amount expressed as a fraction form into a decimal form.

Table 22 – Financial Functions (Continued)

Function	Description
=DOLLARFR(DD, F)	Converts a dollar amount expressed as a decimal form into a fraction form.
=DURATION(S, M, R, Y, F[, B])	The Macauley duration of a security assuming $100 face value.
=EFFECT(NR, NP)	Returns the effective annual interest rate.
=FV(P, R, N)	Future value of an annuity.
=FVSCHEDULE(P, S)	The future value of an initial investment after compounding a series of interest rates.
=INTRATE(S, M, I, R[, B])	The interest rate for a fully invested security.
=IPMT(R, P, NP, PV, FV[, T])	The interest payment for a specific period for an investment based on periodic, constant payments, and a constant interest rate.
=IRR(G, F)	The internal rate of return on an investment. (See also =XIRR and =MIRR.)
=MDURATION(S, M, R, Y, F[, B])	The modified Macauley duration of a security assuming $100 face value.
=MIRR(CF, FR, RR)	The modified internal rate of return for a series of periodic cash flows.
=NOMINAL(ER, NP)	The nominal annual interest rate.
=ODDFPRICE(S, M, I, FC, R, Y, RD, F[, B])	The price per $100 face value of a security with an odd (short or long) first period.
=ODDFYIELD(S, M, I, FC, R, PR, RD, F[, B])	The yield per of a security with an odd (short or long) first period.
=PMT(PV, R, N)	The periodic payment for a loan.
=PPMT(R, P, NP, PV, FV, T)	The payment on the principal for a specific period for an investment based on periodic, constant payments, and a constant interest rate.
=PRICE(S, M, R, Y, RD, F[, B])	The price per $100 face value of a security that pays periodic interest.

Table 22 – Financial Functions (Continued)

Function	Description
=PRICEDISC(S, M, D, RD[, B])	The price per $100 face value of a discounted security.
=PRICEMAT(S, M, I, R, Y[, B])	The price per $100 face value of a security that pays interest at maturity.
=PV(P, R, N)	The present value of an annuity
=RATE(FV, PV, N)	The interest rate required to reach future value FV.
=RECEIVED(S, M, I, D, [, B])	The amount received at maturity for a fully vested security.
=SLN(C, S, L)	The straight-line depreciation allowance.
=SYD(C, S, L, N)	The "sum-of-years-digits" depreciation allowance.
=TBILLEQ(S, M, D)	The bond-equivalent yield (BEY) for a Treasury Bill.
=TBILLYIELD(S, M, D)	The yield on a Treasury bill.
=TERM(P, R, FV)	The number of payment periods for an investment.
=VDB(C, S, L, S, E)	Fixed-declining depreciation allowance between two periods.
=XIRR(G, V, D)	Internal rate of return for a series of cash flows with variable intervals.
=XNPV(R, V, D)	Returns the net present value for a series of cash flows with variable intervals.
=YIELD(S, M, R, PR, RD, F[, B])	Yield of a security that pays periodic interest.
=YIELDMAT(S, M, I, R, PR[, B])	Annual yield of a security which pays interest at maturity.

Date and Time Functions

Table 23 lists the supported date and time functions.

Table 23 – Date and Time Functions

Function	Description
=DATE(Y, M, D)	The date value for year Y, month M, and day D.
=DATEVALUE(S)	The corresponding date value for a given string S.
=DAYS360(S, E)	The number of days between two dates, based on a 30/360 day count system.
=DAY(DT)	The day number in the date/time value DT.
=EDATE(S, M)	The date/time value representing number of months (M) before or after start date (S).
=EOMONTH(S, M)	The date/time value representing the last day of the month M months after S, if M is positive, or M months before if M is negative.
=HOUR(DT)	The hour value (0-23) of date/time value DT.
=MINUTE(DT)	The minute value (0-59) of date/time value DT.
=MONTH(DT)	The number of the month in date/time value DT.
=NETWORKDAYS(S, E[, H])	The number of whole working days, starting at S and going to E, excluding weekends and holidays.
=NOW	The date/time value of the current system date and time.
=SECOND(DT)	The seconds value (0-59) of the date/time value DT.
=TIME(H, M, S)	The time value for hour H, minute M, and second S.
=TIMEVALUE(S)	The corresponding time value for a given string value S.
=TODAY	The date value of the current system date.
=WEEKDAY(D)	The integer representing the day of the week on which the day D falls. 1 is Sunday, 7 is Saturday.
=WORKDAY(S, D[, H])	The day that is D working days after S, if D is positive, or before S, if D is negative, excluding weekends and all holidays specified as dates in range H.

Table 23 – Date and Time Functions (Continued)

Function	Description
=YEAR(DT)	The year value of date/time value DT.
=YEARFRAC(S, E[, B])	The portion of the year represented by the number of days between start date (S) and end date (E).

Miscellaneous Functions

Table 24 lists miscellaneous supported functions.

Table 24 – Miscellaneous Functions

Function	Purpose
=CELLREF(N1, N2)	A reference to the cell in column N1 and row N2.
=CHOOSE(N, ...)	The Nth argument from the list.
=COL(C)	The column address of the cell referenced by C.
=COLS(R)	The number of columns in the specified range R.
=HLOOKUP(X, S, R)	The value of the cell in range S that is R number of rows beneath X.
=INIT(X1, X2)	The first argument on the first recalculation pass and the second argument on all subsequent recalculation passes when Objective Grid for Microsoft .NET is performing iterative calculations.
=INTERP2D(R1, R2, N)	The interpolation value for a 2-dimensional vector.
=INTERP3D(R, X, Y)	The interpolation value for a 3-dimensional vector.
=MATCH(V, R[, T])	The relative position in range R of value V based on positioning criteria T.
=N(R)	The numeric value of the top left cell in range R.
=RANGEREF(N1, N2, N3, N4)	A reference to the range defined by coordinates N1 through N4.
=ROW(C)	The row address of the cell referenced by C.

Table 24 – Miscellaneous Functions (Continued)

Function	Purpose
=ROWS(R)	The number of rows in the specified range R.
=S(R)	The string value of the top left cell in range R.
=VLOOKUP(X, S, C)	The value of the cell in range S that is C number of columns to the right of X.

NOTE >> Some Objective Grid for Microsoft .NET functions return a result that is a range or cell reference. Objective Grid for Microsoft .NET does not include these indirect references in determining the pattern of recalculation. Plan carefully before using these functions.

Embedded Tools

Embedded tools are a set of functions that return data in a matrix, not just the resident cell.

Table 25 – Embedded Tools

Function	Description
=DFT(R)	The Discrete Fourier Transform of the range R.
=EIGEN(M)	The eigenvalues of the matrix M.
=FFT(R)	The Discrete Fourier Transform of the range R using a fast Fourier Transform algorithm.
=FREQUENCY(R, B)	Returns a frequency distribution for values R with a set of intervals B.
=INVDFT(R)	The inverse of the Discrete Fourier Transform of the range R.
=INVERT(M)	The inverse of matrix M.
=INVFFT(R)	The inverse of the Discrete Fourier Transform of the range R using a fast Fourier Transform algorithm.
=LINFIT(X, Y)	The straight line least squares fit. This function is equivalent to =POLYFIT(X, Y, 1).
=LLS(A, Y)	The linear least squares solution X to the overdetermined system of equations AX=Y.
=MMUL(M1, M2)	The product of multiplying matrix M2 by matrix M1.

Table 25 – Embedded Tools (Continued)

Function	Description
=PLS(X, Y, d)	Analyzes the least squares polynomial model Y=P(X), where P is a polynomial of degree d.
=POLYCOEF(X, Y, d)	The least squares coefficients for the polynomial fit Y=P(X), where P is a polynomial of degree d.
=TRANSPOSE(M)	The transpose of matrix M.
=TREND(NX, KX, KY)	The y values for new x values given existing x and y values.

NOTE >> Embedded tools should not be contained within other functions or arithmetic operations in a single formula. For example, the formula =INVERT(=MMUL(A1..C4,F1..I3)) is not allowed. You may, however, copy, move and format embedded tools just as any other function.

Choosing a Grid Type

Objective Grid for Microsoft .NET includes several types of grids:

- **Regular Grid** — You fill the grid with data at startup. The user can edit cells. Changes are stored in the grid.
- **Virtual Grid** — You maintain the data to be displayed in the grid using a data structure or an external file. The grid generally does not store cell contents in memory. You do not fill the grid at startup. Instead, you handle the `GetStyleRowCol` event to supply data on demand. Before it draws the cells on the screen, Grid .NET raises the `GetStyleRowCol` event only for the currently visible cells. If you want the user to be able to edit cells, you need to handle the `StoreStyleRowCol` event and store the changes back into your custom data source. Grid .NET invokes the `StoreStyleRowCol` event whenever a cell is changed. For more information, see Chapter 4, "Virtual Grids."
- **Formula Grid** —You enable the formula engine and fill the grid with data at startup. Cells can contain formulas that reference other cells. When the user edits cells, changes are stored in the formula engine. If a cell that is referenced by a formula is changed, the referring cell is automatically updated.
- **Browser Grid** —A Browser Grid is similar to a Virtual Grid, but the Browser Grid is designed with certain characteristics of record-based data sources in mind. You cannot directly access data in any record. Instead, you loop through the records in order to access the desired record. Then you can read data from the current record and make changes. When the user starts editing a record, changes should only be written back to the recordset when the user navigates to a new record. Therefore, Objective Grid for Microsoft .NET holds changes to the current record in a buffer and flushes this buffer only when the user moves to a new record. Browser Grid also lets the user append rows. The appearance of Browser Grid is similar to MS Access query views.

  Objective Grid for Microsoft .NET provides an ADO.NET Browser Grid implementation.

These guidelines will help you decide what kind of grid to use:

- ***Do you need formula support?***

  Use a Formula Grid.

- ***Does your grid have a large number of rows?***

  Use a Virtual Grid.

- ***Do you maintain data in an external data source or custom data structure?***

  Use an ADO.NET Browser Grid.

- ***Do you want the grid to directly operate on your data?***

  Use Virtual Grid. You do not have to copy all the data at startup into the grid.

- ***Do you want to copy the data into your grid?***

  Use Regular Grid. At startup you fill the grid with your data. The 1stGrid tutorial demonstrates this kind of grid.

Chapter 3

Using Objective Grid for Microsoft .NET

Introduction

This chapter shows examples of the following:

- Resizing grids
- Reading and writing cell values
- Modifying cell attributes
- Sorting rows and columns

Basic Grid Tasks

This section shows you how to change the number of rows and columns in the grid, store data in the grid, and read data from the grid.

NOTE >> The CellValue_CS tutorial, which is located in the Tutorials subdirectory of your Objective Grid for Microsoft .NET installation directory, shows how to write to the grid and read from the grid.

To get started, follow these steps:

1. In Visual Studio, create a new C# "Windows Application" project:
   - **File|New Project|Visual C# projects...**
   - Select **Windows Application**.
   - Name the project and click **OK**. A main form (Form1) is automatically added to the project, and displayed. The form is displayed in design mode. Resize the form as desired.
2. Drop a GridControl onto the empty form. Enlarge the grid control.
3. Build and run the application. The grid is empty and has a default size of 30 rows by 30 columns.
4. Right mouse button click on the form and select **View Code**. Near the top of the file and right after the last "using..." statement add the following line:

```
using Stingray.Grid;
```

Task 1: Changing the Number of Rows and Columns in the Grid

You can change the number of rows and columns in two ways:

- Modify the appropriate grid properties from the form designer (this section)

  This technique sets the initial size of your grid (what appears when you launch your program).

- Call the grid API directly from within your program ("Task 2: Changing the Number of Rows and Columns after Initialization")

  This technique allows you to change the size of the grid any time after program start up.

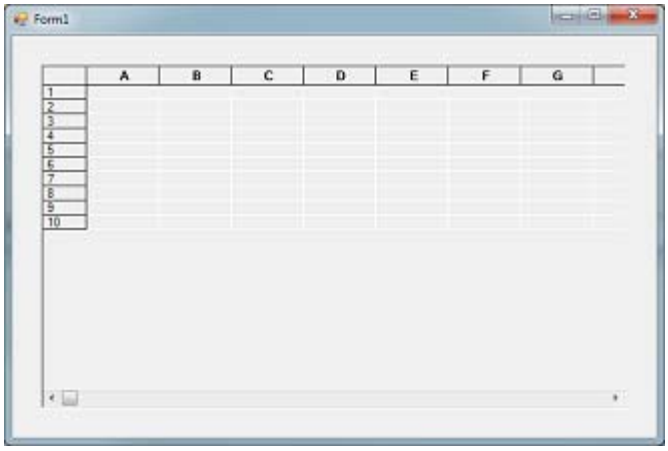
To modify the appropriate grid properties from the form designer:

1. Right-click the form and select **Properties**. Then, right-click the grid and select **Properties**. A separate set of properties appears for each.

2. Modify the grid properties: Find `ColCount` and `RowCount`, and modify their values to 8 and 10, respectively. Press Enter. The grid in the designer shows the new sizes.

3. Build and run the application.

Figure 5 – Grid with Dimensions Set in Property Grid



Task 2: Changing the Number of Rows and Columns after Initialization

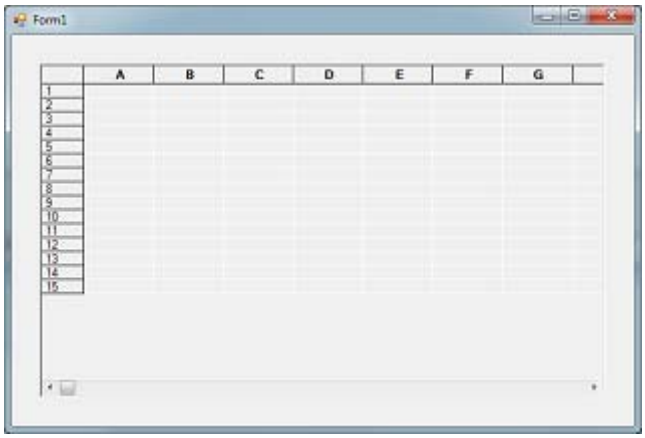
1. In code view, find the constructor (the default name is `Form1()`).

2. After the call to the `InitializeComponent()` method, add the following code:

```
// Resize the grid from the initial size to 15x10
gridControl1.RowCount = 15;
gridControl1.ColCount = 10;
```

3. Build and run the application.

Figure 6 – Grid with Dimensions Set in Code



Task 3: Storing Data in the Grid

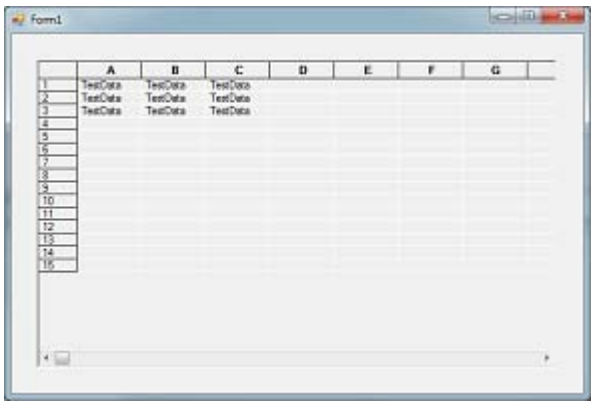
1. Click the form's title bar, and then select the Events button in the properties table (the lightning bolt). A property called `Load` appears under the `Behavior` section.

2. Type *FormLoad* in the value field and press Enter. A class method called `FormLoad()` is created. Add the following code:

```
// Populate several cells
Range r1 = Range.Cells(1, 1, 3, 3);
Style s1 = new Style();
s1.Value = "TestData";
gridControl1.SetStyleRange(r1, s1);
```

3. Build and run the application.

Figure 7 – Data Stored in a Range of Cells

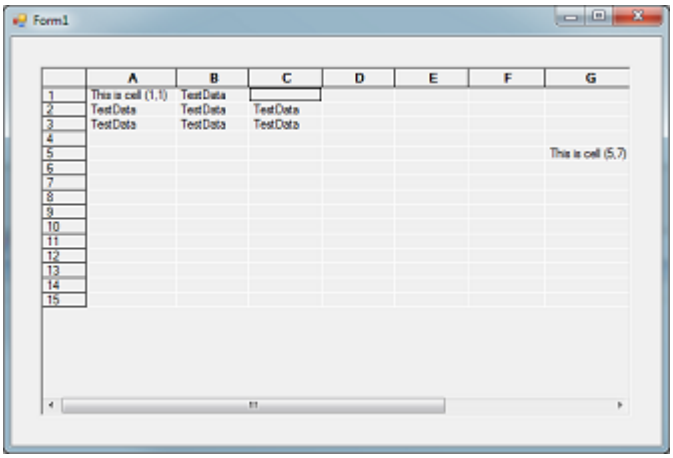


4. Add the following code right after the code fragment you added above:

```
// Manipulate individual cells
gridControl1[1,1].Style.Value = "This is cell (1,1)";
gridControl1[5,7].Style.Value = "This is cell (5,7)";
gridControl1[1,3].ClearValue();
```

5. Build and run the application.

Figure 8 – Data Stored in Individual Cells



When modifying multiple cells, it is a good idea to call `LockUpdate()` to prevent unnecessary redrawing and flickering. To call `LockUpdate()`, write `FormLoad` this way:

```
private void FormLoad(object sender,
                      System.EventArgs e) {
  bool prevLock = gridControl1.LockUpdate(true);

  // Populate several cells
  Range r1 = Range.Cells(1, 1, 3, 3);
  Style s1 = new Style();
  s1.Value = "TestData";
  gridControl1.SetStyleRange(r1, s1);

  // Manipulate individual cells
  gridControl1[1,1].Style.Value = "This is cell (1,1)";
  gridControl1[5,7].Style.Value = "This is cell (5,7)";
  gridControl1[1,3].ClearValue();

  gridControl1.LockUpdate(prevLock);
}
```

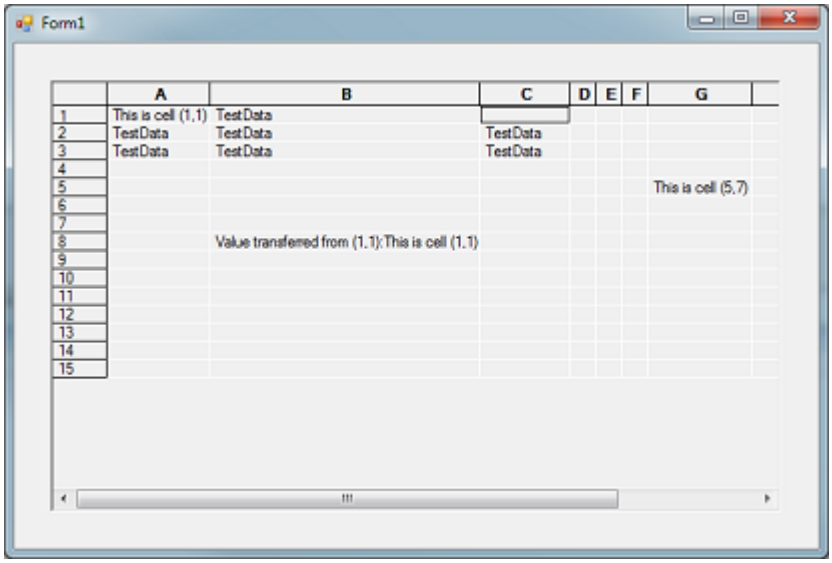
Task 4: Reading Data from the Grid

You can read values from individual cells from the grid using `GetValueRowCol()`:

```
// Copy the contents of (1,1) to another cell
string cellValue = gridControl1.GetValueRowCol(1,1);
gridControl1[8,2].Style.Value = "Value transferred from (1,1):" +
                                cellValue;
```

You cannot use `gridControl1[1,1]` in place of `gridControl1.GetValueRowCol(1,1)`. `gridcontrol[1,1]` represents a ***Cell*** object, not a ***Value***.

Figure 9 – Retrieving Values From a Grid



Cell Types

This section shows you how to modify the attribute of a cell by embedding a control, disable cells, make a cell readonly, cover cells, and merge cells.

NOTE >> The CellAttribute_CS tutorial, which is located in the Tutorials subdirectory of your Objective Grid for Microsoft .NET installation directory, shows how to use styles, covered cells, floating cells, and merged cells.

To get started, follow these steps:

1. Use the steps in "Task 2: Changing the Number of Rows and Columns after Initialization" to set the initial grid size to 5 rows and 5 columns.
2. Turn off the grid lines: Go to the design view for `Form1.cs`. Select the grid, and then find the `Design` section in the properties grid. Set `DrawGrid` to `false`.

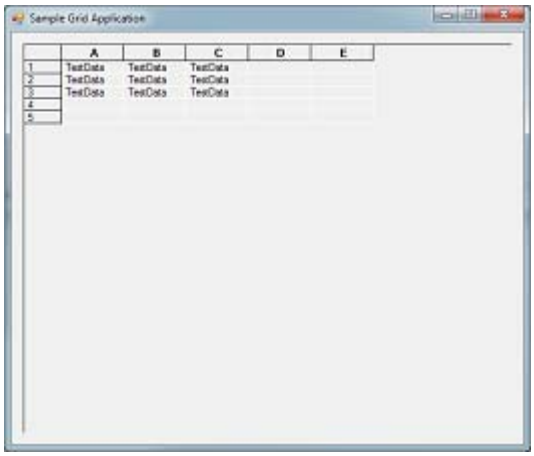
   NOTE >> `DrawGrid` is a property of the form; it is not a Rogue Wave property.

3. Replace the code in `FormLoad()` with the following code:

```
private void FormLoad(object sender,
                      System.EventArgs e) {
  bool prevLock = gridControl1.LockUpdate(true);

  // Populate several cells
  Range r1 = Range.Cells(1, 1, 3, 3);
  Style s1 = new Style();
  s1.Value = "TestData";
  gridControl1.SetStyleRange(r1, s1);

   //
   // ... more stuff to be added here
   //

  gridControl1.LockUpdate(prevLock);
}
```

4. Build and run the application.

Figure 10 – A Small Grid



Task 1: Modify the Attribute of a Cell by Embedding a Control

1. In `FormLoad()`, replace this line of code:

   ```
   // ... more stuff to be added here
   ```

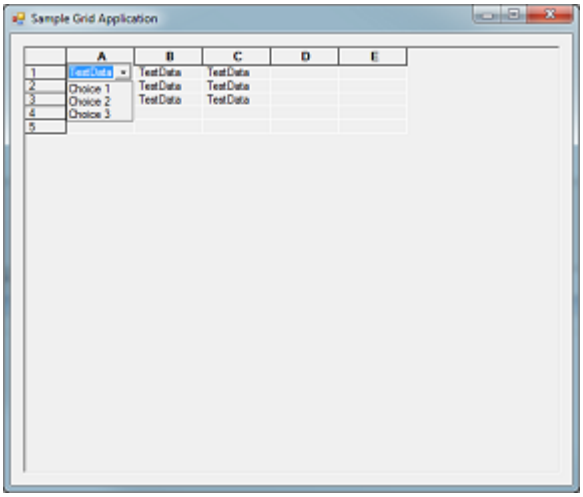
   with the following code:

   ```
   // Change the cell style in cell (1,1) so that it
   // contains a drop down box with 3 choices.
   Style s2 = gridControl1[1,1].Style;
   s2.Control = Stingray.Grid.ControlType.DropDown;
   s2.ChoiceList = "Choice 1\nChoice 2\nChoice 3";
   ```

2. Build and run the application.

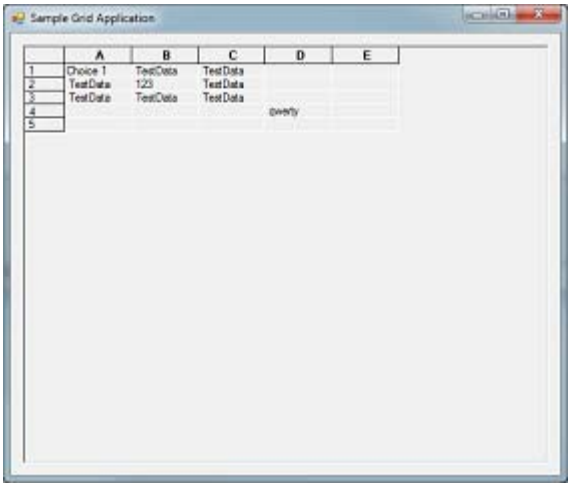
3. Select cell A:1. It turns into a drop-down list from which you can choose `Choice 1`, `Choice 2`, or `Choice 3`, as shown in Figure 11.

Figure 11 – Cell With An Embedded Control



4. Select any cell and change its value.

Figure 12 – Modified Cell



Task 2: Disabling Cells

1. Before the line `gridControl1.LockUpdate(prevLock);` in `FormLoad()`, add the following code:

```
Style s3 = gridControl1[2,2].Style;
s3.Enabled = false;
```

2. Build and run the application.

3. Compare the behavior of cell B2 with the other cells by trying to select it using the mouse or cursor keys. You cannot make cell B2 active.

4. Click cell B2 while pressing the CTRL key, and then press the delete key. The contents of that cell are deleted.

Task 3: Making a Cell Readonly

Making a cell readonly is similar to disabling it. Below this code:

```
Style s3 = gridControl1[2,2].Style;
s3.Enabled = false;
```

add the following code:

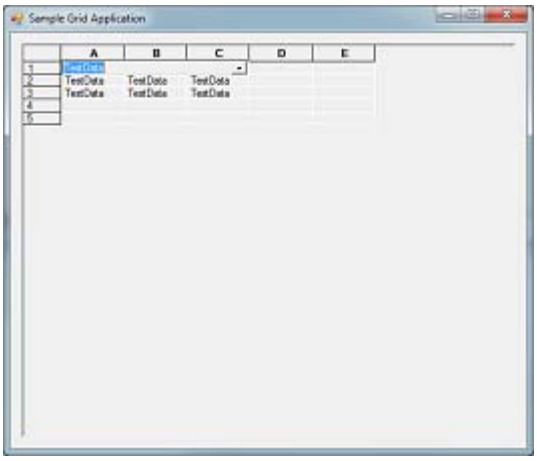
```
s3.ReadOnly = true;
```

Task 4: Covering Cells

Make one cell cover a number of adjacent cells by adding the following code:

```
// Using cell 1,1 to cover cells 1,2 and 1,3
gridControl1.SetCoveredCellsRowCol( 1, 1, 1, 3 );
```

Figure 13 – Covered Cells



Task 5: Merging Cells

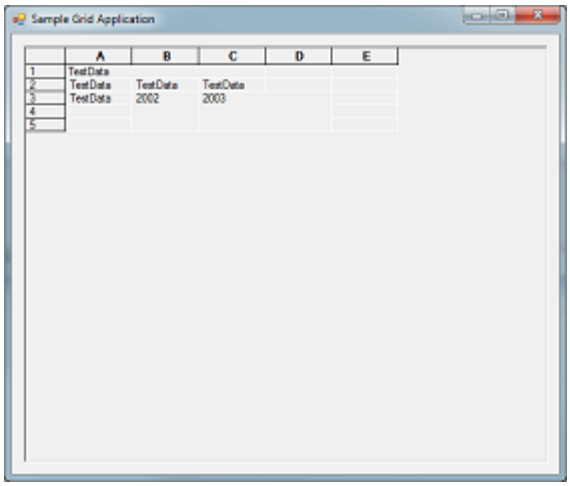
This task requires modifying one of the grid-wide properties.

1. Using the form designer, activate the properties for `gridControl1`.
2. Under the `Behavior` section, locate the `MergeCells` property and set its value to `DelayEval`.
3. Add the following code fragment to the `LoadForm()` method:

```
// Merged Cells
Style sMerged = new Style();
sMerged.MergeCell =
   MergeCellType.BothDirectionsAndSameStyle;
sMerged.Value = "2002";
gridControl1[3,2].Style = sMerged;
gridControl1[4,2].Style = sMerged;
gridControl1[5,2].Style = sMerged;

sMerged.Value = "2003";
gridControl1[3,3].Style = sMerged;
gridControl1[4,3].Style = sMerged;
gridControl1[5,3].Style = sMerged;
gridControl1[3,4].Style = sMerged;
gridControl1[4,4].Style = sMerged;
gridControl1[5,4].Style = sMerged;

sMerged.Dispose();
```

4. Build and run the application.

Figure 14 – Merged Cells



Columns and Rows

This section shows you how to insert new rows and columns, remove rows and columns, hide rows and columns, and freeze rows and columns.

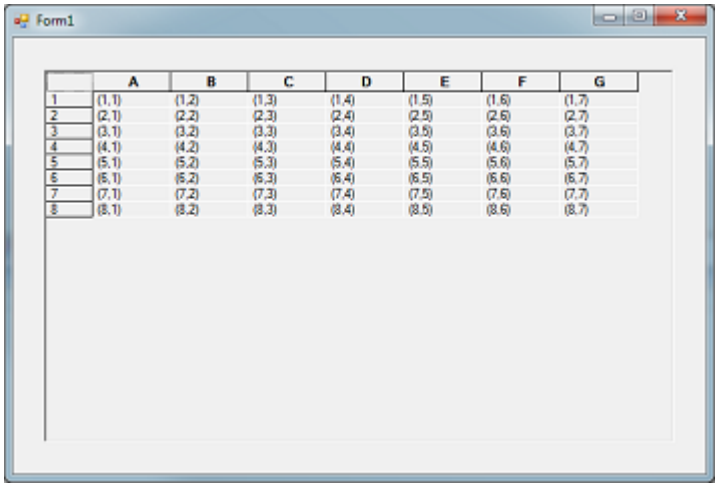
NOTE >> The RowsColumns_CS tutorial, which is located in the Tutorials subdirectory of your Objective Grid for Microsoft .NET installation directory, shows how to hide and freeze rows and columns.

Replace the code in `FormLoad()` with the following code:

```
private void FormLoad(object sender, System.EventArgs e) {
  // Change the size of the grid.
  gridControl1.RowCount = 8;
  gridControl1.ColCount = 7;
  // Fill with data
   for( int i = 1; i <= gridControl1.RowCount; ++i ) {
    for( int j = 1; j <= gridControl1.ColCount; ++j ) {
      gridControl1[i,j].Style.Value = "(" + i + "," + j + ")";
    }
  }
}
```

Build and run the application.

Figure 15 – Adding Rows and Columns In Code



Task 1: Inserting New Rows and Columns

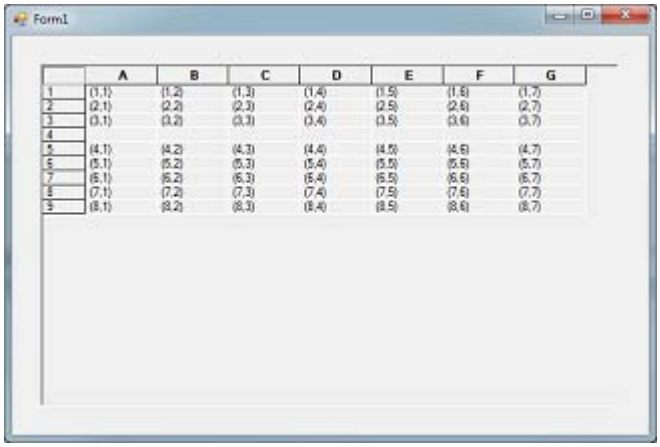
The insertion of a new row or column moves the existing row or column. For example, if you want to add a new row between rows 3 and 4, you specify row 4 as the insertion point. The current row 4 becomes the new row 5.

1. Add the following code to the `LoadForm()` method:

```
// Insert a new row between rows 3 and 4
gridControl1.InsertRows( 4, 1 );
```

2. Build and run the application.

Figure 16 – Inserting a Row

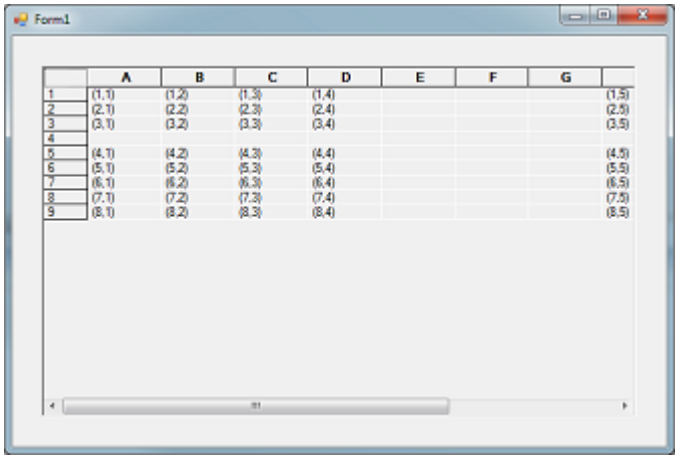


3. Add the following code to the `LoadForm()` method:

```
// Insert three new columns between columns 4 and 5
gridControl1.InsertCols( 5, 3 );
```

4. Build and run the application.

Figure 17 – Inserting Columns



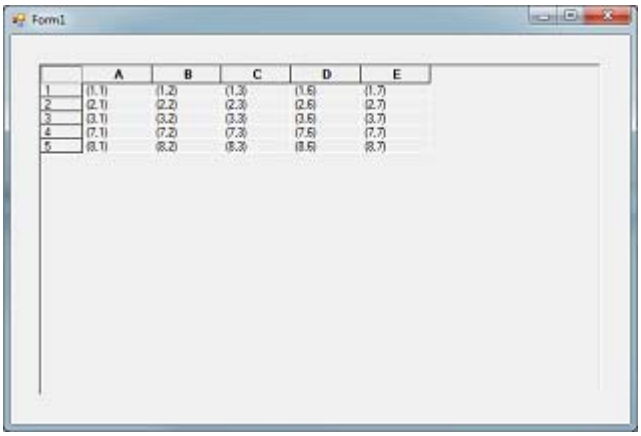
Task 2: Removing Rows and Columns

Removing rows or columns is similar to inserting them. Add the following code to the `LoadForm()` method:

```
// Remove rows 4, 5, 6, and 7. Remember rows after 4
// were displaced by 1 when the new row was added
gridControl1.RemoveRows( 4, 7 );
// Remove columns 4 through 8
gridControl1.RemoveCols( 4, 8 );
```

Build and run the application.

Figure 18 – Removing Rows and Columns



Task 3: Hiding Rows and Columns

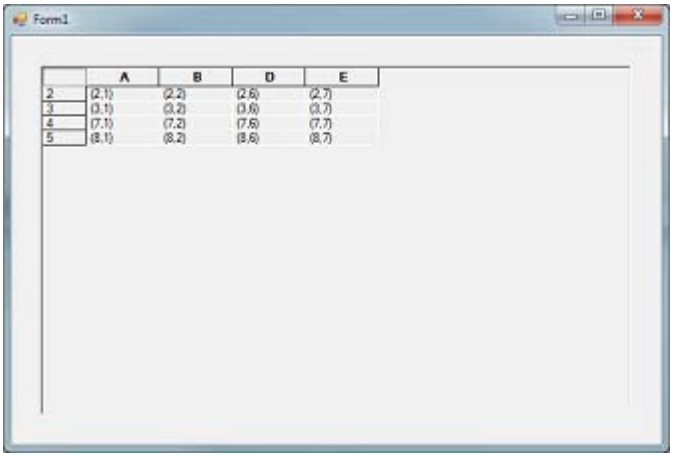
Hiding columns prevents them from being displayed.

1. Add the following code to the `LoadForm()` method:

```
// Hide row 1, and column 3
gridControl1.HideRows( 1, 1, true );
gridControl1.HideCols( 3, 3, true );
```

2. Build and run the application.

Figure 19 – Hiding Rows and Columns



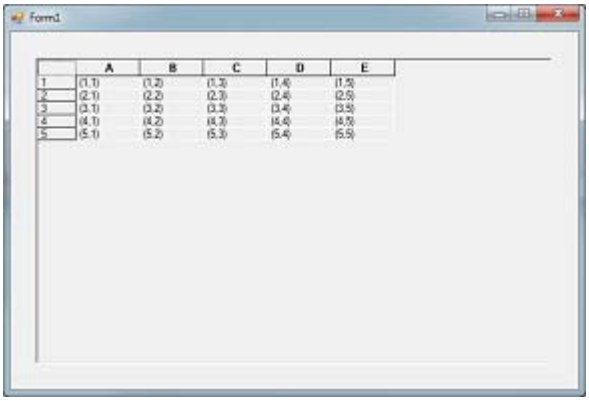
Hidden rows and columns can be made visible again by passing `false` to the `HideRows()` and `HideCols()` methods.

3. An entire range of rows and columns can be hidden by using the ***Range*** class. To illustrate this, set each cell's value to indicate its coordinate within the grid by adding the following code:

```
// Unhide all the rows for the next demonstration
for( int i = 1; i <= gridControl1.RowCount; ++i ) {
  gridControl1.HideRows( i, i, false );
}
for( int i = 1; i <= gridControl1.ColCount; ++i ) {
  gridControl1.HideCols( i, i, false );
}
// Reset the values to indicate rows and columns:
for( int i = 1; i <= gridControl1.RowCount; ++i ) {
  for( int j = 1; j <= gridControl1.ColCount; ++j ) {
    gridControl1[i,j].Style.Value = "(" + i + "," + j + ")";
  }
}
```

4. Build and run the application.

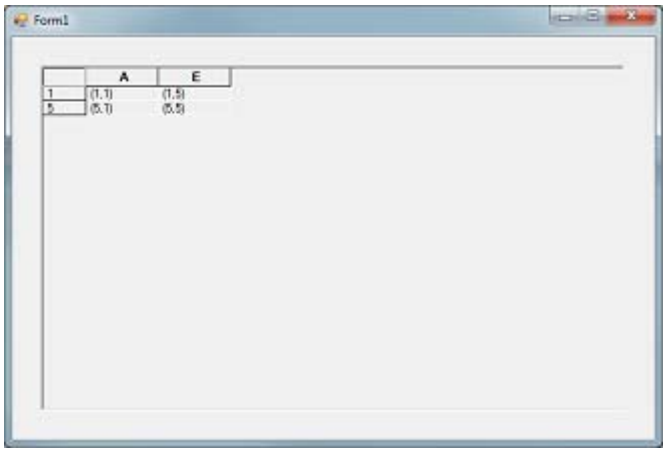
Figure 20 – All Cells Visible



5. Add code to hide cells B2 through D4:

```
// Hide the cells B2 to D4
Range r1 = Range.Cells(2, 2, 4, 4);
gridControl1.HideRange( r1, true );
```

Figure 21 – A Hidden Range



Task 4: Freezing Rows and Columns

When rows or columns are frozen, they do not scroll with the rest of the grid.

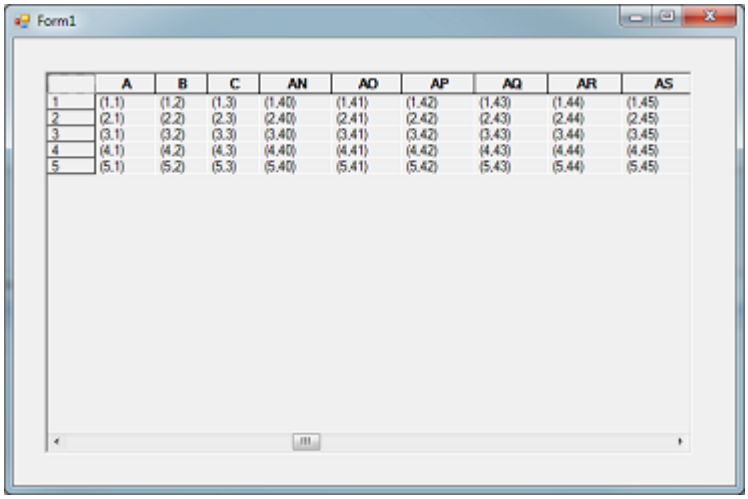
Replace the `FormLoad()` method with the following code:

```
private void FormLoad(object sender, System.EventArgs e)
{
  // Change the size of the grid.
  gridControl1.RowCount = 5;
```

```
    // To test the action of the grid with a frozen
    // column, add enough cells to cause a horizontal
    // scroll bar to appear
    gridControl1.ColCount = 100;
    fillGrid();
    gridControl1.FrozenCols = 3;
  }

  private void fillGrid()

  // fillGrid fills in all cells in the grid with data
  // reflecting the row and column indices of each cell.
  // This shows what happens to the grid when some
  // columns are frozen, and the view is scrolled
  // horizontally to reveal the rightmost columns.

  {
    Style s1;
    for( int i = 1; i <= gridControl1.RowCount; ++i )
    {
      for( int j = 1; j <= gridControl1.ColCount; ++j )
      {
        s1 = gridControl1[i,j].Style;
        s1.Value = "(" + i + "," + j + ")";
      }
    }
  }
```

Figure 22 shows the state of the grid after scrolling over several columns. The first three columns remain fixed, but the fourth and succeeding columns numbers increase.

Figure 22 – Grid With Frozen Columns



Cell and Window Coordinates

This section shows you how to compute the cell coordinate from a given window point.

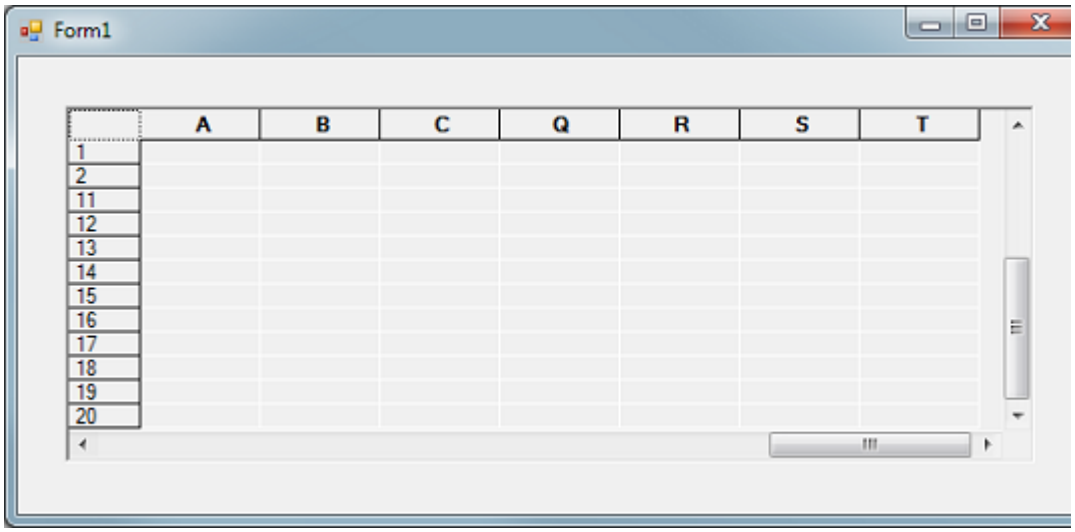
NOTE >> The CellWindowCoord_CS tutorial, which is located in the Tutorials subdirectory of your Objective Grid for Microsoft .NET installation directory, shows how to use cell and window coordinates.

1. If needed, in the designer, resize the grid so that no more than 15 rows and columns are visible.
2. Replace the `FormLoad()` method with the following code:

```
private void LoadForm(object sender, System.EventArgs e) {
// Change the size of the grid.
gridControl1.RowCount = 20;
gridControl1.ColCount = 20;
// Freeze the first two rows and first three columns.
gridControl1.FrozenCols = 3;
gridControl1.FrozenRows = 2;
}
```

3. Build and run the application.
4. Scroll the rows and columns all the way to the end. With the scrollbars at their maximum positions, the row indices start at 1, 2, and then skip to 11.

Figure 23 – Frozen Rows and Columns



5. Add the following function to `Form1.cs`:

```
private void recalc() {
  int x1, x2;
  int y1, y2;
  for( int i = 1; i <= gridControl1.RowCount; ++i ) {
    for( int j = 1; j <= gridControl1.ColCount; ++j ) {
      x1 = gridControl1.ConvertClientCol( j );
      y1 = gridControl1.ConvertClientRow( i );
      x2 = gridControl1.ConvertCol( j );
      y2 = gridControl1.ConvertRow( i );
      gridControl1[i,j].Style.Value = "(" + x1 + "," + y1 + ")=>("
                                          + x2 + "," + y2 + ")";
    }
  }
}
```

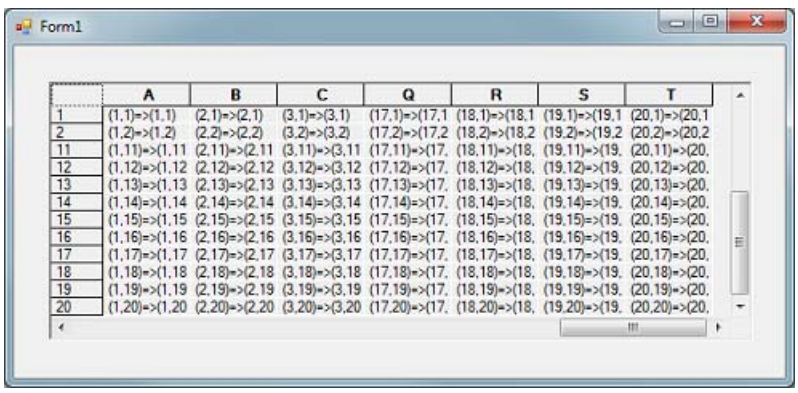
6. In `FormLoad()`, add a call to `recalc()`:

```
recalc();
```

7. Build and run the application.

8. Set the scrollbars to their maximums, and widen the last two columns to show the full cell text.

Figure 24 – Coordinates Displayed



The next step is adding a little bit of user interaction to trigger a recalculation.

1. Drag a main menu from the Toolbox onto the form.

2. In the field labeled "Type here" at the top of the form, enter *Recalculate*.

3. In the properties form, click the event button and enter *recalculate_click* as the value for the `click` property.

4. In the code view of the form, locate and modify the `recalculate_click()` method:

```
private void recalculate_click(object sender,
```

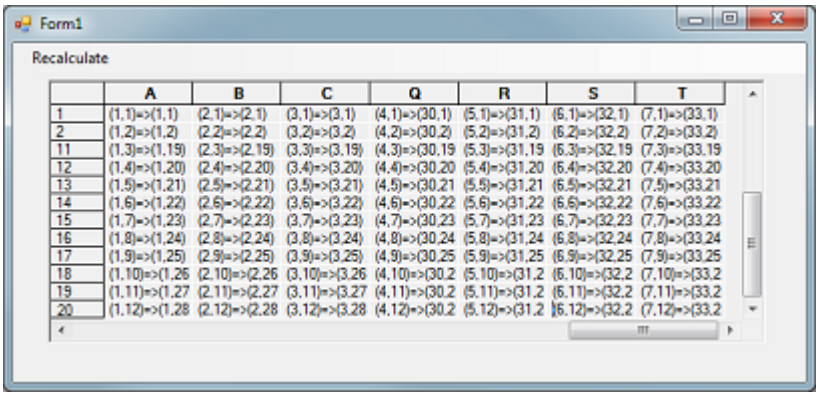
```
        System.EventArgs e) {
        recalc();
    }
```

This calls the method that recalculates and fills the cells on your command.

5. Build and run the application.

6. Set the scrollbars to their maximums, and then click the Recalculate menu. The cells take on new values, except for those in the frozen region (1,1)...(3,2).

Figure 25 – Grid With Recalculated Coordinates



Sorting a Grid

Objective Grid for Microsoft .NET allows you to sort a grid by either columns or rows. This section demonstrates how to enable sorting. See also (“Sorting” for information on how you can customize Objective Grid for Microsoft .NET to enable other sorting-related features.)

NOTE >> The Sort_CS tutorial, which is located in the Tutorials subdirectory of your Objective Grid for Microsoft .NET installation directory, shows how to support sorting when users double-click row or column headers.

Replace the `FormLoad()` method with the following code:

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

gridControl1.RowCount = 10;
gridControl1.ColCount = 10;

  Random r = new Random();
  for( int i = 1; i <= gridControl1.RowCount; ++i ) {
    for( int j = 1; j <= gridControl1.ColCount; ++j ) {
      gridControl1[i,j].Style.Value = "" + r.Next(30);
    }
  }
}
```

1. Build and run the application.

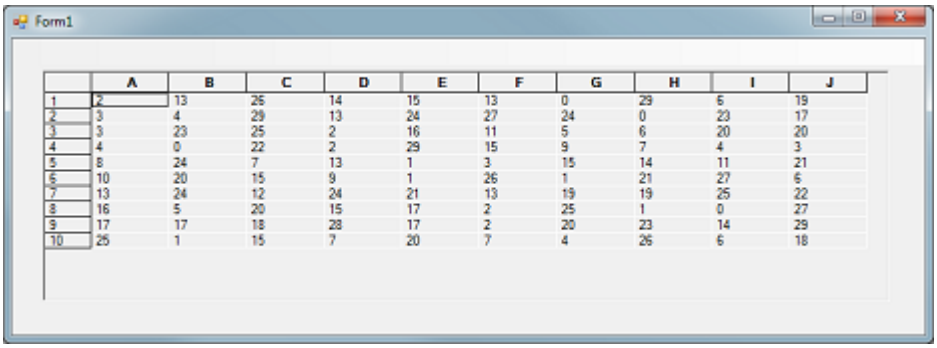
Figure 26 – Grid With Random Integers

	A	B	C	D	E	F	G	H	I	J
1	23	18	14	19	1	28	3	23	22	28
2	25	6	21	18	20	22	6	14	24	1
3	3	0	8	3	18	29	2	7	21	20
4	24	13	16	27	18	21	15	21	1	22
5	11	2	8	4	24	5	23	1	6	5
6	10	13	7	1	1	23	10	11	0	1
7	26	25	2	1	3	11	20	23	29	28
8	6	14	12	2	20	0	21	3	8	13
9	6	0	0	15	9	11	5	15	26	25
10	11	4	21	29	25	15	0	7	1	13

2. In the Form Designer, click the grid and view the grid properties.
3. Locate the `EnableHorizontalSorting` and `EnableVerticalSorting` properties in the `Data` property group. Set these properties to `true`.
4. Build and run the application.

5. Double-click the column header labeled "A." The rows are sorted in ascending order based on each row's value in column "A." Figure 27 shows the result.

Figure 27 – Sorted Column



	A	B	C	D	E	F	G	H	I	J
1	2	13	26	14	15	13	0	29	6	19
2	3	4	29	13	24	27	24	0	23	17
3	3	23	25	2	16	11	5	6	20	20
4	4	0	22	2	29	15	9	7	4	3
5	8	24	7	13	1	3	15	14	11	21
6	10	20	15	9	1	26	1	21	27	6
7	13	24	12	24	21	13	19	19	25	22
8	16	5	20	15	17	2	25	1	0	27
9	17	17	18	28	17	2	20	23	14	29
10	25	1	15	7	20	7	4	26	6	18

6. Double-click the column header again to resort the values in descending order.

You can programmatically control whether or not the rows or columns can be sorted. Assuming the grid object is called `gridControl1`:

```
gridControl1.Param.EnableHorizontalSorting = true;
gridControl1.Param.EnableVerticalSorting = false;
```

This code fragments allows rows to be sorted and disallows sorting for columns.

Extending .NET Functionality

Objective Grid for Microsoft .NET is a wrapper over the Objective Grid MFC libraries. It is easy to customize the libraries to implement the functionality you need. This section provides several examples of customization.

Customizing Behavior

You can customize the behavior of your applications in a number of ways. This section includes some examples.

Accepting Numeric Data Only

You can limit the data in a cell to accept only numeric data. For example:

```
this.gridControl1.ModifyCell +=
         new Stingray.Grid.ModifyCellEventHandler(this.gridControl1_ModifyCell);

private void gridControl1_ModifyCell(object sender,
                                     Stingray.Grid.ModifyCellEventArgs e)
{
     string strValue = gridControl1[e.Row,e.Col].Style.Value;
     string strText = gridControl1[e.Row,e.Col].ControlText;
     if(!IsNumber(strText))
           gridControl1[e.Row,e.Col].Style.Value = strValue;
     else
           gridControl1[e.Row,e.Col].Style.Value = strText;
}

//this function is from http://www.c-sharpcorner.com/3/RegExpPSD.asp
private bool IsNumber(String strNumber)
{
     Regex objNotNumberPattern=new Regex("[^0-9.-]");
     Regex objTwoDotPattern=new Regex("[0-9]*[.][0-9]*[.][0-9]*");
     Regex objTwoMinusPattern=new Regex("[0-9]*[-][0-9]*[-][0-9]*");
     String strValidRealPattern="^([-]|[.]|[-.]|[0-9])[0-9]*[.]*[0-9]+$";
     String strValidIntegerPattern="^([-]|[0-9])[0-9]*$";
     Regex objNumberPattern =new Regex("(" + strValidRealPattern +")
                                        |(" + strValidIntegerPattern + ")");
return !objNotNumberPattern.IsMatch(strNumber) &&
objTwoDotPattern.IsMatch(strNumber) &&
objTwoMinusPattern.IsMatch(strNumber) &&
 objNumberPattern.IsMatch(strNumber);
}
```

Highlighting a Value in a Cell

To highlight the value in a cell, add the following function to the file `GridControl.h`:

```
void SetSel(int nStartChar, int nEndChar);
```

In the source file `GridControl.cpp`, add:

```
void GridControl::SetSel(int nStartChar, int nEndChar)
{
   ROWCOL nCurRow, nCurCol;
   m_pCtrl->GetCurrentCell(&nCurRow, &nCurCol);
   if (m_pCtrl->GetCurrentCellControl()->
                  IsKindOf(RUNTIME_CLASS( CGXEditControl)))
   if (((CGXEditControl*)m_pCtrl->GetCurrentCellControl())->OnStartEditing())
   {
       ((CGXEditControl*)m_pCtrl->GetCurrentCellControl())->
                                 SetActive(TRUE);
       ((CGXEditControl*)m_pCtrl->GetCurrentCellControl())->
                                 SetSel(nStartChar,nEndChar);
       ((CGXEditControl*)m_pCtrl->GetCurrentCellControl())->Refresh();
   }
}
```

After rebuilding the assemblies, you can use code similar to the following:

```
gridControl1[2,2].Style.Value = "Hello";
gridControl1.CurrentCell = gridControl1[2,2];
gridControl1.SetSel(1,3);
```

Advancing the Cursor from the Last Column to the Next Row's First Column

To advance a cursor from the last column in one row, to the first column of the next when the user presses either the right arrow, enter, or tab key, you can create a custom event handler. For example:

```
this.gridControl1.ProcessKeys
+= new Stingray.Grid.ProcessKeysEventHandler
                                    (this.gridControl1_ProcessKeys);

private void gridControl1_ProcessKeys(object sender,
                     Stingray.Grid.ProcessKeysEventArgs e)
{

     if(( e.Char == 39 || e.Char == 9 || e.Char == 13)
     && e.Message == 256
     && gridControl1.CurrentCell.Col == gridControl1.ColCount)
     {
          int col = gridControl1.CurrentCell.Col;
          int row = gridControl1.CurrentCell.Row;
          gridControl1.LockUpdate();
```

```
        if(gridControl1.MoveCurrentCell(Stingray.Grid.DirectionType.Down))
            {
                gridControl1.LockUpdate(false);
                gridControl1.MoveCurrentCell(
                                Stingray.Grid.DirectionType.MostLeft);
                gridControl1.RedrawRowCol(row,col);
                e.Handled = true;
            }
            else
                gridControl1.LockUpdate(false);
        }

    }
```

In most cases, the above works. However, when a cell is a combo box (for instance, `style.Control = ControlType.DropDownList;`), the enter/tab/left arrow/right arrow keys won't advance to the next row. Here is a workaround:

```
    protected override System.Boolean ProcessCmdKey (ref Message m ,
                                                     Keys keyData )
    {
        if(keyData == Keys.Tab || keyData == Keys.Enter ||
                                                     keyData == Keys.Right
           &&gridControl1.CurrentCell.Style.Control == ControlType.DropDownList)
            {
                bool bRight = gridControl1.MoveCurrentCell(
                                        Stingray.Grid.DirectionType.Right);
                if(bRight)
                        return true;

                if(!bRight && gridControl1.CurrentCell.Col ==
                                        gridControl1.ColCount)
                {
                        int col = gridControl1.CurrentCell.Col;
                        int row = gridControl1.CurrentCell.Row;
                        gridControl1.LockUpdate();
        if(gridControl1.MoveCurrentCell(Stingray.Grid.DirectionType.Down))
                        {
                                gridControl1.LockUpdate(false);
        gridControl1.MoveCurrentCell(Stingray.Grid.DirectionType.MostLeft);
                                gridControl1.RedrawRowCol(row,col);
                                gridControl1.RedrawRowCol(row-1,col);
                                return true;
                        }
                        gridControl1.LockUpdate(false);
                }
            }
    return base.ProcessCmdKey(ref m, keyData);
    }
```

Extending the Last Column to Infinity When Scrolling Horizontally

You can extend the last column to infinity during horitontal scrolling using this code:

```
private void gridControl1_HorizontalScroll(object sender,
                  System.Windows.Forms.ScrollEventArgs e)
{
  int nColCount = gridControl1.ColCount;
  Rectangle GridRect = gridControl1.GetGridRect();
  int nRight = gridControl1.CalcRectFromRowColEx(1,nColCount).Right;
  gridControl1.SetColWidth(nColCount,nColCount,
  gridControl1.GetColWidth(nColCount)+ GridRect.Right-nRight);
}
```

Setting the Max and Min Date in a DateTime Control

Here is an example for setting the Max and Min date values in a `DateTime` control:

```
Style style = new Style();
style.Control = Stingray.Grid.ControlType.DateTime;
style.set_UserAttribute(UserAttributeID.DATEMIN,"1/1/2000");
style.set_UserAttribute(UserAttributeID.DATEMAX,"1/1/2005");
style.set_UserAttribute(UserAttributeID.DATEVALIDMODE,"2");
gridControl1.SetStyleRange(Range.Cell(1,1),style);
```

Making the Current Cell Inactive

To deactivate the current cell in MFC version of Grid, we can use the following:

```
SetCurrentCell(GX_INVALID, GX_INVALID);
```

We can implement the same feature in Grid for .NET, changing the source code in file `gridcontrol.h`:

```
[ Browsable(false) ]
        __property Stingray::Grid::Cell *get_Cell(Int32 nR, Int32 nC)
        {
          if(!(nR == -1 && nC == -1))
          validateRowCol(nR, nC);

          return new Stingray::Grid::Cell(nR, nC, this, m_pCtrl);
        }
```

After rebuilding the libraries, you can use the following in your program code:

```
gridControl1.CurrentCell = gridControl1[-1,-1];
```

Retrieving the Date from a DateTime Control

You can use the function `GetValueRowCol()` to return the date from a cell with a `DateTime` control, rather than retrieving just the numeric value.

For example, the following function returns a `System.DateTime` object:

```
private DateTime GetDateFromCell(int nRow, int nCol)
{
string Value = gridControl1.GetValueRowCol(nRow,nCol);
DateTime celldt;
if(Value !="")
{
int nValue = Convert.ToInt32(Value);
DateTime initdt = new DateTime(1899,12,30,12,0,0);
TimeSpan ts = new TimeSpan(nValue,0,0,0);
return celldt = initdt + ts;
}
else
return celldt = DateTime.Today;
}
```

Tabbing to a Grid for .NET Control on a Form

When tabbing through all the controls on a Windows Form, the Grid Control may not receive focus (i.e. may be skipped) even though the Grid Control's TabIndex and TabStop properties are set correctly. The following source code works around this issue in the Windows Form application. All changes below refer to the application's Windows Form `*.cs` file containing the Grid Control along with other .NET controls.

In the ***System.Windows.Forms.Form*** derived, public class, add the following private, initialized boolean variable:

```
private bool bEnableGridTabFocus = false;
```

In the Windows Form Designer generated class private function `InitializeComponent()`, under the Grid Control section add the following event handler:

```
this.gridControl1.ProcessKeys += new Stingray.Grid.ProcessKeysEventHandler(
                                        this.gridControl1_ProcessKeys);
```

After the Windows Form Designer generated class region, add the following protected override and private delegate:

```
protected override System.Boolean ProcessCmdKey(ref Message m, Keys keyData)
{
    if (keyData == Keys.Tab || (keyData == (Keys.Shift | Keys.Tab)))
    {
        if (this.GetNextControl(Control.FromHandle(m.HWnd), true) ==
                                                gridControl1)
        {
            bEnableGridTabFocus = true;
            return gridControl1.Focus();
        }
    }
        return base.ProcessCmdKey(ref m, keyData);
}

private void gridControl1_ProcessKeys(object sender,
                                      Stingray.Grid.ProcessKeysEventArgs e)
{
    if ((e.Char == 9 || e.Char == 65536) && !bEnableGridTabFocus)
    {
        GetNextControl(gridControl1, true).Focus();
        e.Handled = true;
    }
    else if (bEnableGridTabFocus)
        bEnableGridTabFocus = false;
}
```

Rebuild the application to test the tab and shift+tab traversal and focus of each indexed control on the Windows Form.

Adding a Cut Event Handler for Use Within a Virtual Grid

To extract characters from a cell within a Virtual Grid, add a Cut event handler within the Grid Control as shown below:

```
this.gridControl1.Cut +=
            new Stingray.Grid.CutEventHandler(this.gridControl1_Cut);

 private void gridControl1_Cut (object sender,System.EventArgs e)
 {
          gridControl1.SetSel(0,-1);
 }
```

For information on the `setSel()` function, see "Highlighting a Value in a Cell."

Loading OGL Files

You can load OGL files in an application that uses the Objective Grid for Microsoft .NET component, like so:

```
private void InitializeComponent()
{
  //fix for OGL
  this.gridControl1 = new Stingray.Grid.GridControl("..\\..\\Layout1.OGL");

((System.ComponentModel.ISupportInitialize)(this.gridControl1)).BeginInit();
  this.SuspendLayout();
  //
  // gridControl1
  //
  this.gridControl1.Anchor = (((System.Windows.Forms.AnchorStyles.Top
                              | System.Windows.Forms.AnchorStyles.Bottom)
                              | System.Windows.Forms.AnchorStyles.Left)
                              | System.Windows.Forms.AnchorStyles.Right);
  this.gridControl1.AutoScroll = true;
  this.gridControl1.BorderStyle = System.Windows.Forms.BorderStyle.Fixed3D;

  //fix for OGL
  this.gridControl1.FloatCells = Stingray.Grid.FloatCellsMode.EvalOnDisplay;
  this.gridControl1.EnableUndo = true;
  this.gridControl1.Size = new System.Drawing.Size(544, 408);
  this.gridControl1.TabIndex = 0;

  //
  // GridDocument
  //
  this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
  this.ClientSize = new System.Drawing.Size(544, 413);
  this.Controls.AddRange(new System.Windows.Forms.Control[]
                        {this.gridControl1});

((System.ComponentModel.ISupportInitialize)(this.gridControl1)).EndInit();
  this.ResumeLayout(false);
}
```

This feature is implemented in `<stingray-installdir>\Objective Grid for Microsoft .NET\Tutorials\1stGrid\Step 2`.

Customizing the Display

This section discusses ways in which you can customize how your application is displayed to users.

Displaying Ellipses for Values Greater than a Column Width

You can display ellipses if the text in a column is greater than the column's width. To enable the use of ellipses:

- Set the EllipseType in `Properties\Current Cell\Style`
- Set `WrapText` to `FALSE`.

For example:

```
private void gridControl1_GridInitialized(object sender, System.EventArgs e)
    {
       Style style = new Style();
       style.Value = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaa";
       style.WrapText = false;
       style.EllipseType = EllipseType.PoundEllipse;
       Range range = Range.Rows(2,5);
       gridControl1.SetStyleRange(range,style);
    }
```

Customizing the Display of Wide Grids that Flow off the Page

For very wide grids in which right-hand columns are not fully displayed, default Objective Grid for Microsoft .NET behavior is to shift the entire grid to the left when a user clicks in a far right-hand column, in order to allow its display.

You can disable this behavior by returning `FALSE` in `CGXDynRegularGrid<T>::ScrollCellInView,` like this:

```
template<class T>
BOOL CGXDynRegularGrid<T>::ScrollCellInView(ROWCOL nRow, ROWCOL nCol,
                                            UINT flags /*= GX_UPDATENOW*/,

BOOL bFloatCell /*= FALSE*/)
{
        //return (m_bShowHierarchy == FALSE) ?
        //T::ScrollCellInView(nRow, nCol, flags, bFloatCell) :
        //CGXExRegularGrid<T>::ScrollCellInView(nRow, nCol, flags, bFloatCell);
        return FALSE;
}
```

Finally, rebuild GridControl.

Displaying the Time

By default, the `style.control DateTime` displays only the date, but you can also easily display the time. The format of the `DateTime` control is set by the `UserAttribute` (see the GridApp sample provided for an MFC Grid) as in the following code snippet:

```
private void gridControl1_GridInitialized(object sender, System.EventArgs e)
{
        Style s = new Style();
        s.Control = Stingray.Grid.ControlType.DateTime;
        s.set_UserAttribute(Stingray.Grid
                                    .UserAttributeID.CUSTOMFORMAT, "hh:mm");
        s.Value = DateTime.Today.ToString();
        gridControl1[1,1].Style = s;
}
```

Changing the Color of a Cell's Top Border

To change the color of a cell's top border, edit the code in the file `Stingray Studio\Objective Grid for Microsoft.NET\Solutions\GridControl\Style\Style.h`, to add the code block between the comments, like this:

```
__property virtual void set_Borders( CellBorders *value )
        {
          AFX_MANAGE_STATE(AfxGetStaticModuleState());
          if (value->IsAllBordersSame())
          {
            cgxStyle_->SetBorders(gxBorderAll,
                    Stingray::Grid::_marshal::CGXPen(value->Left));
          }
          else
          {
            cgxStyle_->SetBorders(gxBorderLeft,
                    Stingray::Grid::_marshal::CGXPen(value->Left));
            cgxStyle_->SetBorders(gxBorderTop,
                    Stingray::Grid::_marshal::CGXPen(value->Top));
            cgxStyle_->SetBorders(gxBorderRight,
                    Stingray::Grid::_marshal::CGXPen(value->Right));
            cgxStyle_->SetBorders(gxBorderBottom,
                    Stingray::Grid::_marshal::CGXPen(value->Bottom));
          }
        }
    }
    //////////////// insert this code block ////////////////
   else if (value->Left == NULL || value->Top==NULL || value->Right ==
                                            NULL ||value->Bottom == NULL)
{
      if (value->Left !=NULL)
            cgxStyle_->SetBorders(gxBorderLeft,
```

```
            Stingray::Grid::_marshal::CGXPen(value->Left));
      else if (value->Top!=NULL)
            cgxStyle_->SetBorders(gxBorderTop,
            Stingray::Grid::_marshal::CGXPen(value->Top));
      else if (value->Right != NULL)
            cgxStyle_->SetBorders(gxBorderRight,
            Stingray::Grid::_marshal::CGXPen(value->Right));
      else if (value->Bottom !=NULL)
            cgxStyle_->SetBorders(gxBorderBottom,
            Stingray::Grid::_marshal::CGXPen(value->Bottom));
    }
  ////////////////////////////////////////////////////////////
    else {
            cgxStyle_->SetBorders(gxBorderLeft,
            Stingray::Grid::_marshal::CGXPen(value->Left));
            cgxStyle_->SetBorders(gxBorderTop,
            Stingray::Grid::_marshal::CGXPen(value->Top));
            cgxStyle_->SetBorders(gxBorderRight,
            Stingray::Grid::_marshal::CGXPen(value->Right));
            cgxStyle_->SetBorders(gxBorderBottom,
            Stingray::Grid::_marshal::CGXPen(value->Bottom));
          }

   }
```

After rebuilding the assemblies, you can do the following:

```
gridControl1[e.Row, e.Col].Style.Borders =
new Stingray.Grid.CellBorders( null,null,null,
new Stingray.Grid.OGPen(BorderDashStyle.Dot, Color.Yellow, 1));
```

Setting an Interior Color for a Button Control

Set an interior color for a button control like this:

```
Style s = new Style();
s.Control = ControlType.Button;
s.ChoiceList = "Set Color";
s.set_UserAttribute(UserAttributeID.BUTTONFACECOLOR,
               ColorTranslator.ToWin32(Color.White).ToString());
gridControl1.SetStyleRange(Range.Cell(3,1),s,CellModifyType.Override);
```

Hiding Columns

To prevent the user from resizing hidden columns, use code like this:

```
private void gridControl1_StoreHideCol(object sender,
                       Stingray.Grid.StoreHideColEventArgs e)
{
```

```
    if(gridControl1.IsColHidden(e.Col)&& !e.Hide)
       e.Hide = true;
}
```

Setting the Font in a Range of Cells to Bold

To set the font in a range of cells to bold:

```
Style style = new Style();
style.TextFont = new OGFont("Arial", 10, 0, true,false,false,false);
gridControl1.SetStyleRange(Range.Cols(2,3), style);
```

Making a Combobox Visible in an Inactive Cell

If you wish to make combobox button visible in inactive cell, add this code to the file `GridControl.h`:

```
public:
void SetComboButtonVisible (int ID, bool Visible)

and in file GridControl.cpp

void GridControl::SetComboButtonVisible (int ID, bool bVisible)
{
 ((CGXComboBoxWnd*)m_pCtrl->GetRegisteredControl(ID))
                          -> m_bInactiveDrawButton = bVisible;

}
```

After rebuilding the assemblies, you can use this code like so:

```
gridControl1.SetComboButtonVisible( GX_IDS_CTRL_CBS_DROPDOWN, true);
```

Formatting a Numeric Cell Using Currency Control and UserAttributeIDs

UserAttributeIDs are enumeration types that marshall the MFC Grid style types used by the Objective Grid for Microsoft .NET grid control for the formatting of numeric cells.

The Objective Grid for Microsoft .NET UserAttributeIDs can be found in `<stingray-install-dir>\Objective Grid for Microsoft .NET\Solutions\GridControl\Constants\OGEnumsAndConstants.h`.

The marshalling of these enumeration types can be found in `<stingray-install-dir>\Objective Grid for Microsoft .NET\Solutions\GridControl\Style\Style.h` and `Style.cpp`.

Objective Grid for Microsoft .NET control provides a Currency control, `Stingray.Grid.ControlType.Currency`, which is flexible and easy to use.

Below are explanations and usage examples of some of these .NET UserAttributeIDs as used by the Grid Control's Currency control. Each item contains code snippets from the larger code block below illustrating the use of UserAttributeIDs to format numeric cells in various ways.

The comments in the larger code block reference the code snippet by item number for readability and ease of use.

```
private void Form1_Load(object sender, System.EventArgs e)
    {
          Style NumStyle = new Style();
          NumStyle.Control = Stingray.Grid.ControlType.Currency;

          // See #1, “UserAttributeID.CURUMDECIMALS: Setting the Number
          // of Decimal Places for Numbers.”
          NumStyle.set_UserAttribute(UserAttributeID.CURNUMDECIMALS,"4");

          // See #2, “UserAttributeID.CURNEGFORMAT: Changing
          // the Display of Negative Numeric Values.”
          NumStyle.set_UserAttribute(UserAttributeID.CURNEGFORMAT, "4");

          // See #3, “UserAttributeID.CURSEP:Setting the Thousand
         // Separator Delimiter Character.”
          NumStyle.set_UserAttribute(UserAttributeID.CURSEP,",.");

          // See #4, “UserAttributeID.CURMON: Constraining the Input
          // of Valid Numeric Data Only.”
          NumStyle.set_UserAttribute(UserAttributeID.CURMON, "0");

          gridControl2.SetStyleRange(Range.Cell(1,1),NumStyle);

          // See #1, “Required Event Handlers.”
          gridControl2[1,1].Style.set_UserAttribute(
                                          UserAttributeID.CURNUMFRACT,"2");

          gridControl2[1,1].Style.Value = "1234.1234";
    }

    private void gridControl2_EndEditing(object sender,
                                         Stingray.Grid.EndEditingEventArgs e)
    {
          Style style = gridControl2[1,1].Style;

          // See #1, “Required Event Handlers.”
          style.set_UserAttribute(UserAttributeID.CURNUMFRACT,"2");

          gridControl2.SetStyleRange(Range.Cell(1,1), style);
    }

    private void gridControl2_StartEditing(object sender,
```

```
                                                Stingray.Grid.StartEditingEventArgs e)
{
      Style style = gridControl2[1,1].Style;

      // See #1, “Required Event Handlers.”
      style.set_UserAttribute(UserAttributeID.CURNUMFRACT,"4");

      gridControl2.SetStyleRange(Range.Cell(1,1),style);
}
```

1. **UserAttributeID.CURUMDECIMALS: Setting the Number of Decimal Places for Numbers**

   The end user should be able to change the number of decimal places at runtime.

   For example, if the end user enters a value of 1000.00002 and the current format setting for the decimal place is set to 2, then the numeric value should be displayed as 1000.00, while the grid's cell maintains the actual value of 1000.00002.

```
NumStyle.set_UserAttribute(UserAttributeID.CURNUMDECIMALS,"4");
```

Required Event Handlers

Note that event handlers for Start and End editing are required for correct runtime behavior. These event handlers are implemented as shown:

```
. . .

          gridControl2[1,1].Style.set_UserAttribute(
                                    UserAttributeID.CURNUMFRACT,"2");

      . . .
      }

      private void gridControl2_EndEditing(object sender,

Stingray.Grid.EndEditingEventArgs e)
      {
          Style style = gridControl2[1,1].Style;

          style.set_UserAttribute(UserAttributeID.CURNUMFRACT,"2");

            gridControl2.SetStyleRange(Range.Cell(1,1), style);
      }

      private void gridControl2_StartEditing(object sender,

Stingray.Grid.StartEditingEventArgs e)
      {
          Style style = gridControl2[1,1].Style;
```

```
            style.set_UserAttribute(UserAttributeID.CURNUMFRACT,"4");

              gridControl2.SetStyleRange(Range.Cell(1,1),style);
        }
```

2. **UserAttributeID.CURNEGFORMAT:Changing the Display of Negative Numeric Values**

   The end user should be able to change how negative numbers are displayed at runtime.

   For example, if an end user enters -22.22, the numeric value should be displayed as (22.22), while the grid's cell maintains the actual value of -22.22.

   ```
   NumStyle.set_UserAttribute(UserAttributeID.CURNEGFORMAT, "4");
   ```

3. **UserAttributeID.CURSEP: Setting the Thousand Separator Delimiter Character**

   The end user should be able to change the number of decimal places at run time.

   For example, if the end user enters a value of 10000, the numeric value should be displayed as 10,000, while the grid's cell maintains the actual value of 10000.

   ```
   NumStyle.set_UserAttribute(UserAttributeID.CURSEP,",.");
   ```

4. **UserAttributeID.CURMON: Constraining the Input of Valid Numeric Data Only**

   The end user should be able to enter only valid numeric data in any given cell at runtime.

   For example, the end user should be able to enter only numeric values in any given cell or range of cells with this attribute.

   If the end user tries to enter some other non-numeric character, the cell should ignore that character.

   ```
   NumStyle.set_UserAttribute(UserAttributeID.CURMON, "0");
   ```

   Please refer to the section, UserAttributeID Enumeration, for a complete list of attributes and as listed in the Objective Grid for Microsoft® .NET® Reference Guide.

   Please refer to the *MFC Objective Grid's User's Guide and Reference Guide* for details of the style types called from the MFC Objective Grid and marshalled by the Objective Grid for Microsoft .NET grid control.

Matching Background Colors in a Grid

You may change the background color of all cells in the grid. But to change the background color of the whole grid, so that the background color of the cells and the remaining area of the grid matches, you must make additional changes.

In the MFC Grid, use the following function:

```
CGXProperties::SetColor(int nIndex, COLORREF rgbColor)
```

and then set nIndex, like this: `nIndex = GX_COLOR_BACKGROUND`.

In Objective Grid for Microsoft .NET, add the following code into `public __gc class OGProperties` in the `OGProperties.h` file, located in the `..\Solutions\GridControl\Param` folder, as shown here:

```
__property void set_BkColor(Color value)
{
gxProp->SetColor(GX_COLOR_BACKGROUND,ColorTranslator::ToWin32(value));
}
```

After rebuilding the assemblies, you can access this property from your C# code. After these changes, you can use the following code in your project:

```
this.gridControl1.BackColor = System.Drawing.Color.Azure;  // cells
this.gridControl1.Properties.BkColor = System.Drawing.Color.Azure;
                                                        // out of cells
```

Printing

This section describes how to set print margins and enable color printing.

Setting Print Margins

In MFC Grid, use the following functions to set the print margins (left, right, top, bottom):

```
CGXProperties::SetDistances(int top, int bottom);
CGXProperties::SetMargins(int top, int left, int bottom, int right);
```

To set print margins with Objective Grid for Microsoft .NET using VC#, add the following code into `public __gc class OGProperties` in the `OGProperties.h` file, located in the `..\Solutions\GridControl\Param` directory:

```
void SetDistances(int top, int bottom)
{
gxProp->SetDistances( top, bottom);
}

void SetMargins(int top, int left, int bottom, int right)
{
gxProp->SetMargins(top, left, bottom, right);
}
```

Rebuild the solution from the folder, `..\Solutions\GridControl`. With the new assemblies, you will be able to call these functions in your project.

Enabling Color Printing

To enable/disable color printing for Objective Grid using VC#, add the following code into `public __gc class OGProperties` in the `OGProperties.h` file, located in the `..\Solutions\GridControl\Param` directory:

```
__property void set_BlackWhite(bool b)
{
gxProp-> SetBlackWhite(BOOL b);
}
__property bool get_BlackWhite()
{
return gxProp->GetPrintBlackWhite();
}
```

After rebuilding the assemblies, you will be able to access this property from your C# code.

Grid Line Display and Printing

Even if you are only displaying some cell grid lines, Objective Grid for Microsoft .NET may show grid lines for all cells in Print Preview and print them all as well. Exposing the following functions from the MFC base code to C# (similar to the approach demonstrated in "Setting Print Margins") should resolve this problem:

```
CGXProperties::SetPrintHorzLines
CGXProperties::SetPrintVertLines
```

Sorting

You can customize the behavior and display when sorting, discussed in this section.

Marking the Sort Column

You can mark the sorting column with a small triangle in the header, using the following code:

```
private void gridControl1_LButtonDblClkRowCol(object sender,
                            Stingray.Grid.LButtonDblClkRowColEventArgs e)
{
      if(e.Row ==0)
      {
            Style style = gridControl1[0,e.Col].Style;
            style.TextFont = new OGFont("Wingdings 3", 10, 0);
            style.Value = "q"; //"p";
            style.TextColor = System.Drawing.Color.Blue;
      }
}
```

Sorting with Empty Cells

The code below sorts the values of a column when that column is double-clicked:

```
gridControl1.Param.EnableHorizontalSorting = true;
```

```
gridControl1.Param.EnableVerticalSorting = false;
```

For columns that lack a value in any row, you need to edit the code in the file `gxDynRegularGrid.h` and then rebuild the control, or attempts to sort the column result in a debug assert.

For example:

```
template<class T>
void CGXDynRegularGrid<T>::OnGetSortRowsKey(ROWCOL nRow,
                            const CGXSortInfoArray& sortInfo,
                            CStringArray& keys)
{
       m_bShowHierarchy == FALSE) ?
              T::OnGetSortRowsKey(nRow, sortInfo, keys) :
              CGXExRegularGrid<T>::OnGetSortRowsKey(nRow, sortInfo, keys);
              if(keys[0].IsEmpty())
                         keys[0] = _T("zzzzzz");
}
```

Memory and Performance Issues

This section provides tips and troubleshooting when dealing with memory or performance issues.

Performance Issues when Displaying Numbers

There are a few ways to set values in Objective Grid for Microsoft .NET: using property `Style.Value` or functions `SetValueRange()` and `SetStyleRange()`. The test in this section was used in the 1stGrid Tutorial sample to compare performance and speed for these methods.

Using `LockUpdate(true)` and `LockUpdate(false)` around the code of interest increases performance.

Add this code to the application's WinForm to perform benchmark testing:

```
private int nTicks;

private void timer1_Tick(object sender, System.EventArgs e)
{
    for(int nRow = 1; nRow<=10; nRow++)
        for(int nCol =1; nCol<=5; nCol++)
            gridControl1[nRow,nCol].Style.Value = nTicks.ToString();
    nTicks++;
}

private void Form1_Load(object sender, System.EventArgs e)
{
    this.timer1.Start();
    this.timer1.Interval = 100;
    this.timer1.Tick += new EventHandler(timer1_Tick);
}
```

A menu option or button control may be added to the application's WinForm to perform benchmark tests.

For example, the button click event below was added to show how to benchmark test three different approaches when setting numeric values in the grid.

```
private void button1_Click(object sender, System.EventArgs e)
{
    //gridControl1.LockUpdate(true);

    // Benchmark test of numbers set with Style.Value
    DateTime dtStart = DateTime.Now;
    for(int n =1; n <=1000; n++)
    {
        gridControl1[1,1].Style.Value = n.ToString();
    }
    DateTime dtStop = DateTime.Now;
    TimeSpan duration = dtStop - dtStart;
    Console.WriteLine(duration);

    // Benchmark test of numbers set with SetValueRange()
    Style style = new Style();
    dtStart = DateTime.Now;
    for(int n =1; n <=1000; n++)
    {
       style.Value = n.ToString();
       gridControl1.SetValueRange(Range.Cell(1,1), style);
    }
    dtStop = DateTime.Now;
    duration = dtStop - dtStart;
    Console.WriteLine(duration);

    // Benchmark test of numbers set with SetStyleRange()
    dtStart = DateTime.Now;
    for(int n =1; n <=1000; n++)
    {
       style.Value = n.ToString();
       gridControl1.SetStyleRange(Range.Cell(1,1), style);
    }
    dtStop = DateTime.Now;
    duration = dtStop - dtStart;
    Console.WriteLine(duration);

    //gridControl1.LockUpdate(false);
}
```

Test Results:

Benchmark test results ***without*** the use of LockUpdate(true) and LockUpdate(false):

```
00:00:01.6718750 // Duration of numbers set with Style.Value
```

```
00:00:01.5312500 // Duration of numbers set with SetValueRange()
00:00:01.5468750 // Duration of numbers set with SetStyleRange()
```

Benchmark test results ***with*** the use of LockUpdate(true) and LockUpdate(false):

```
00:00:00.4218750 // Duration of numbers set with Style.Value
00:00:00.2968750 // Duration of numbers set with SetValueRange()
00:00:00.2812500 // Duration of numbers set with SetValueRange()
```

Troubleshooting

This section includes possible issues you may encounter and methods for resolving them.

Releasing Memory

This section provides a workaround if an application appears to increase in memory usage.

Add these two two lines of code anywhere in your application where memory should be released.

```
HANDLE h = GetCurrentProcess();
SetProcessWorkingSetSize( h, -1, -1 );
```

A similar approach works with C#, where you can add this line at the top of the application's `Form.cs` file:

```
using System.Runtime.InteropServices;
```

Then add this code block to the application's WinForm class, public class `Form1 : System.Windows.Forms.Form`:

```
[DllImport("kernel32.dll", SetLastError=true)]
          static extern int SetProcessWorkingSetSize (
                                        int hProcess,
                                        int dwMinimumWorkingSetSize,
                                        int dwMaximumWorkingSetSize);
          [DllImport("kernel32.dll", SetLastError=true)]
          static extern int GetCurrentProcess();
```

Finally, call these two lines of code anywhere in your application where memory should be released.

```
int h = GetCurrentProcess();
SetProcessWorkingSetSize( h, -1, -1 );
```

Chapter 4

Virtual Grids

Background

Data can be stored and displayed in two ways with Objective Grid for Microsoft .NET GridControl objects:

- First, the GridControl properties `RowCount` and `ColCount` and the GridControl methods `SetStyleRange` and `SetValueRange` can be used to set the size of the grid and to store data directly to the grid.

  With this method, the grid data is stored directly in the GridControl's parameter object. The size and content of the grid is managed entirely by the GridControl and its parameter object.

- Second, the `GetStyleRowCol` and `StoreStyleRowCol` events can be used to store and retrieve grid data from your own data structure or structures.

  With this method, you are responsible for managing, to some degree, the size and content of the grid.

For Objective Grid for Microsoft .NET, "Virtual Grid" refers to the second use case (or some hybrid of both use cases).

Reasons for Using Virtual Grids

A virtual grid (or virtual mode) is ideal when the data to be manipulated with the grid is actually persisted in other data structures or files, such as a database.

Virtual mode can also improve the performance of your grid applications. If the number of rows and columns in your grid is large and your grid takes a long time to populate with data, virtual might perform better because the grid needs data only for the cells that are currently in view, not for every row. In other words, you supply values from your data structure for display in the grid only when the grid requests them. Grid requests are completed by overriding `OnGetStyleRowCol` or by attaching a delegate to the `GetStyleRowCol` event.

When data that is displayed in the grid is ready to be stored, GridControl invokes `OnStoreStyleRowCol`. You can override this method or attach a delegate to the `StoreStyleRowCol` event to store the data in your own data structures. For readonly grids, you would not have to handle the `StoreStyleRowCol` event.

Using Virtual Grids

For basic virtual grid behavior, you must handle these events:

- `GetStyleRowCol`
- `StoreStyleRowCol`

You can also make the following choices:

- You can choose to control the size of the grid in virtual mode by overriding the `GetRowCount` and `GetColCount` methods.
- You can choose to derive a new class from ***GridControl*** and override `OnGetStyleRowCol`, `OnStoreStyleRowCol`, and optionally, `GetRowCount` and `GetColCount`.

  If you do not want to control the size of the grid in virtual mode, you can simply attach delegates to the `GetStyleRowCol` and `StoreStyleRowCol` events.
- You can choose how much of the grid's style information you want to store to and retrieve from your own data structures. Many virtual grid applications choose to store only the value property of the cell style (the cell's text). However, you are free to store as much of the additional style information as is appropriate for your application.

Advanced Virtual Grid Topics

Only the Value property from the style is stored, and the grid does not store any non-default styles. The grid manages its own size.

As mentioned in "Using Virtual Grids", Objective Grid for Microsoft .NET allows for some flexibility in the implementation of virtual grids. The following changes could be made to the sample to allow for greater control over the virtual grid:

- More than just the cell value could be stored in the hash table. If desired, the entire style could be stored. User-specific data could also be stored for each cell.
- For greater control over the virtual grid, a derived grid class could be created. This would allow the user to control the size of the grid by overriding `GetRowCount` and `GetColCount` to return specific values.

NOTE >> When overriding `GetRowCount` and `GetColCount`, you should be aware of the following: `GetRowCount()` and `GetColCount()` are used extensively to validate grid values. In addition, these methods may be called on every mouse move event. It is not unusual for `GetRowCount()` to be called thousands of times in a short period of time. Because `GetRowCount()` and `GetColCount()` are called quite often, your overrides of these methods are not an appropriate place to do extensive calculations.

- The `StoreStyleRowColEventArgs` object supplied to `StoreStyleRowCol` event handlers has two properties that can be used to further customize the behavior of virtual grids.
  - The `Stored` property—This property can be set to `true` to indicate that all style information that needs to be stored has been stored by your event handler. When set, it prevents ***GridControl*** event handling, which stores the style to the grid. If the `SetStyleRange` is invoked with style information other than cell value information, that style information would not be stored in either the grid or the data structure. If a hybrid approach is desired, where the grid would store some font or color information, for example, then the `Stored` property should *not* be set to `true`.
  - The `CanStore` property—This property indicates whether or not the style information for the cell could be stored. For example, you may choose to perform some data validation at this time. If the data is not valid and cannot be stored, this value should be set to `false`. If `false`, the current cell cannot be changed, and the user must supply a valid value for the cell.

Chapter 5

.NET Cell Controls

Introduction

Objective Grid for Microsoft .NET allows any .NET control to be easily embedded and used as a grid cell editor. This allows controls from the .NET runtime environment, as well as custom .NET controls, to be used in grid cells.

Overview

The following classes, methods, and properties are required for using .NET controls in grid cells:

- ***CellControl***—This class represents a custom cell control and wraps any .NET control for use in a grid cell.
- `GridControl.RegisterControl`—This method accepts a single argument, a `System.Windows.Forms.Control`-derived object, and returns an instance of a `CellControl` that wraps the .NET control.
- `Style.CustomControl`—This property accepts a `CellControl` object created by `GridControl.RegisterControl`. The specified cell control object can then be used in grid cells using `GridControl.SetStyleRange`.

The following C# code segment illustrates how to use a .NET `NumericUpDown` as a cell editor in column 2 of a grid control. This should be executed only during or after the invocation of the `GridControl.GridInitialized` event.

```
// Instantiate a NumericUpDown control
System.Windows.Forms.NumericUpDown upDown =
new System.Windows.Forms.NumericUpDown();

// Configure the control
upDown.Minimum = -10;
upDown.Maximum = 10;
upDown.BorderStyle = BorderStyle.None;

// Instantiate a Objective Grid for Microsoft .NET Style object, and
// assign its CustomControl property to the registered CellControl
Style upDownCell = new Style();
upDownCell.CustomControl = this.GridControl1.RegisterControl(upDown);
upDownCell.Value = “0”;

// Use SetStyleRange to use the control in column 2
this.GridControl1.SetStyleRange(Range.Col(2), upDownCell);
```

Features

The following sections describe features of the .NET cell control functionality in Objective Grid for Microsoft .NET.

.NET Controls Can Be Used as Cell Editors

Any properly functioning .NET control can be used as a cell editor in Objective Grid for Microsoft .NET. You can use the .NET controls shipped with Microsoft Visual Studio .NET or create your own custom controls for use as cell editors. You can also embed a Objective Grid for Microsoft .NET GridControl object in grid cells.

Using a .NET control as a cell editor requires the following four steps:

1. Create an instance of the .NET control that you want to use as a grid cell editor.
2. Register the control with the ***GridControl*** instance that will use it as a cell editor. Do this using the `GridControl.RegisterControl` method. It returns a CellControl instance.
3. Assign the resulting ***CellControl*** instance to the `CustomControl` property of a ***Style*** object.
4. Apply the ***Style*** to the ***GridControl*** using `SetStyleRange`.

Memory Resources Are Conserved

To conserve memory resources, only one instance of any .NET control is maintained for use as a grid cell editor. For example, if a .NET `TextBox` control is used as a cell editor for the entire grid table, only one instance of the `TextBox` control is maintained. It is automatically positioned, sized, initialized, and drawn in each cell as it becomes the current cell.

Style Manipulation Code Is Usually Not Needed

By default, the `Text` property for your .NET control is set to the `Style.Value` setting for the cell when it is initialized as the current cell. When cell editing is completed, the `Text` property for your control is stored back to the grid's `Style.Value` property for the cell. This, in many cases, eliminates the need to write any style manipulation code.

Cell Control Behavior Can Be Monitored and Customized

The behavior of your .NET cell control can be monitored and customized using the various properties and events associated with the ***CellControl*** class. The following are examples of the kinds of behaviors that can be configured using ***CellControl*** properties and events:

- Using the `UseCellStyle` property, the .NET ***CellControl*** can be displayed with the font, background color, and text color stored in the grid's style for the cell. Conversely, the colors and font assigned to the .NET control can be used regardless of the stored style for the grid cell.
- You can choose to be notified each time your .NET ***CellControl*** is drawn. This allows you do use GDI+ to draw inactive cells as desired. For example, a bitmap can be drawn in inactive combo-box cells to show that a combo box control is used in the cell.
- You can choose to be notified of various editing events involving the cell control, including `StartEditing`, `CancelEditing`, `EndEditing`.
- You can choose to be notified when the cell control is initialized as the current cell and when the cell contents need to be stored to the grid. By default, the .NET control's Text property is used to initialize the cell and store the cell's data. However, for more complicated controls, such as cells with an embedded ***GridControl*** object, you can use these events to write more complicated initialization and storage functionality.

Cell Editing Events Can Be Monitored

You can choose to handle cell editing events by monitoring the ***CellControl*** events directly or by monitoring the corresponding ***GridControl*** events. For example, the `StartEditing` event for a cell control can be monitored using the `CellControl.StartEditing` event or via the `GridControl.StartEditing` event.

Limitations

The following sections describe limitations to using .NET controls with Objective Grid for Microsoft .NET.

Some Controls Cannot Be Sized Vertically

Some of the .NET controls that are supplied with Visual Studio .NET do not always size themselves as requested. For example, the controls `NumericUpDown`, `TextBox`, and `ComboBox` cannot be sized vertically. Their vertical size is determined by the font used in the control, rather than by the setting of the `Size` property. These .NET controls may not fill the entire vertical area of a grid cell when used as grid cell editors. Similarly, they may occupy more vertical area than the cell in which they are embedded. To overcome this limitation, you may choose to write your own custom controls for these features.

CellControl Behavior Must Be Customized Using Delegates

The current design does not permit derivation from the ***CellControl*** class to customize the behavior of .NET cell controls. Currently, customizing the behavior of ***CellControl*** objects must be achieved by attaching delegates to the events associated with ***CellControl*** instances.

The .NET Control Can Be Drawn in Only One Cell at a Time

The .NET control can be drawn in only one cell at a time because only one instance of a .NET control is used for entire ranges of grid cells. As such, it is not immediately obvious that a particular type of control is used in an inactive cell that uses a .NET control. To overcome this limitation, attach a delegate to the ***CellControl.DrawCellControl*** event and draw inactive (non-current) cells as desired using GDI+.

Supports Registration of a Maximum of 32 .NET Controls

A single ***GridControl*** object supports the registration of a maximum of 32 .NET controls.

Using .NET Cell Controls

The following C# code shows the simplest use-case for using a .NET control in a grid cell. It illustrates the use of a .NET `TextBox` in grid cell 1, 1.

```
// Instantiate a Style object. Set the CustomControl property
// to a registered TextBox control. Then, set the resulting style
// for grid cell 1,1.
Style s = new Style();
s.CustomControl = this.GridControl1.RegisterControl(new TextBox);
this.GridControl1.SetStyleRange(Range.Cell(1, 1), s);
```

You can dramatically customize the behavior of .NET cell controls using the following mechanisms:

- You can set properties of the `TextBox` before registering it with the `GridControl`. You can also attach delegates to the .NET control for event handling purposes. Finally, you can write your own custom .NET control for use as a cell editor.
- You can customize the ***CellControl*** instance returned from `GridControl.RegisterControl`. ***CellControl*** exposes a number of public properties and events that can be used to customize the behavior of the cell controls. For more detailed information about these properties and events, see ***CellControl*** in the *Objective Grid for Microsoft® .NET® Reference Guide*.

Table 26 lists and describes the public properties available for the ***CellControl*** class.

Table 26 – Cell Control Properties

Property	Description
`DrawInactiveCell`	Gets or sets a value that is `true` if you want the grid to draw inactive cell contents. When you are drawing inactive cells, set the value to `false`.
`Grid`	Gets the ***GridControl*** with which this custom control is associated.
`ProcessHorizontalArrows`	Gets or sets a value that is `true` if your custom control processes horizontal arrow key events. If `false`, the right and left arrow keys move the current cell in the corresponding direction.
`ProcessVerticalArrows`	Gets or sets a value that is `true` if your custom control processes vertical arrow key events. If `false`, the up and down arrow keys move the current grid cell in the corresponding direction.
`UseCellStyle`	Gets or sets a value that indicates whether the cell style from the grid should be applied to the custom control. If `true`, the font, background color, and text color stored in the grid are applied to the control when it is initialized as the current cell. If `false`, the settings applied to the .NET control are used to display the control.

Table 27 lists and describes the public events available for the ***CellControl*** class. For each of these events, a corresponding `On` event handler raises the event. Each `On` event handler is invoked by the grid, and in turn, the event handler raises the event, invoking any delegates that you may have attached to the event.

For more detailed information describing the default processing associated with each of these events, see ***CellControl*** in the *Objective Grid for Microsoft® .NET® Reference Guide*.

Table 27 – CellControl Events

Event	Description
`CanceledEditing`	This event is raised when the current cell's contents are canceled and the control has been reinitialized.
`CancelEditing`	This event is raised when the user has pressed ESC on the current cell.
`DrawCellControl`	This event is raised each time this ***CellControl*** is drawn, which depends mostly upon whether the cell is the current cell. The event arguments for this event indicate whether the cell is the current cell. At this time, this event is intended to notify when cells are drawn and to allow the end user to implement custom drawing logic for inactive cell controls.
`EndEditing`	This event is raised when the user confirms the cell's data, and the data is valid. You are given the opportunity to indicate whether the data are accepted via the event arguments.
`HideCurrentCell`	This event is raised when the ***CellControl*** should be hidden.
`InitCurrentCell`	This event is raised by ***GridControl*** when the intrinsic state (for example, row, column, and Style*) of the control for the current cell needs to be initialized. It is invoked when the current cell has moved and the ***CellControl*** is being initialized as the new current cell.
`LeftCell`	This event is raised after the current cell is deactivated. At the time this event is raised, the grid does not have a current cell. You can attach a delegate to this event if you want to control the refreshing of the current cell after it is deactivated.
`ModifyCell`	This event is raised when the user modifies the cell's contents.
`RefreshCurrentCell`	This event is raised when the cell needs to be refreshed.
`ResetCurrentCell`	This event is raised when the grid resets the intrinsic state of the control. This happens when the current cell is moving.
`StartEditing`	This event is raised when the user starts editing the cell by pressing a key or moving the mouse.

Table 27 – CellControl Events

Event	Description
`StoreCurrentCell`	This event is raised to store the cell's data to the grid. For efficiency, this method is raised only if the cell contents are modified. For your convenience, the cell is marked as modified as part of the event handling for the `StartEditing` event.
`ValidateCell`	This event is raised to validate the contents of a grid cell.

Chapter 6

GridTabControl Control

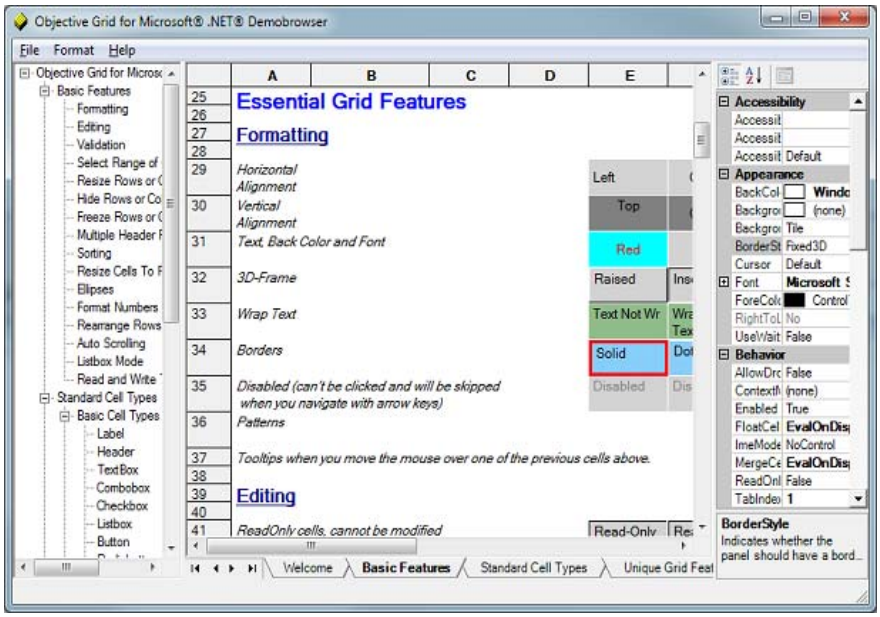
Introduction

From the end-user perspective, the functionality of the ***GridTabControl*** control is like the functionality of a workbook in Microsoft Excel. The ***GridTabControl*** control switches between several grid tabs. It displays a tab-beam control on the bottom of the control. Objective Grid for Microsoft .NET objects and any other objects derived from the ***Control*** class controls can be added as tab items.

Overview

The idea of ***GridTabControl*** is to provide the Excel-like workbook interface to the application developers and end-users. The main differences between the ***GridTabControl*** and .NET Framework's ***TabControl*** class (besides the MS Excel look) is that ***TabControl*** can contain only objects of ***TabPage*** class as tab items. The ***GridTabControl***, although designed mainly as a container for ***GridControl***-derived objects can also contain objects of any ***Control***-derived class.

Figure 28 – The GridTabControl in action



The functionality of the control is simple. After creating the control, you can call `InsertTab` to insert several tabs, or, at design time, you can drop the ***GridTabControl*** on the form, and then drop any other controls on its surface. For each control dropped on the ***GridTabControl***'s surface, a new tab is created.

You can click the scrolling buttons on the left side to scroll the display of tabs to the left or right by one tab. To select a sheet, click on the tab. The clicked tab is highlighted, and the control associated with it becomes visible and activated.

By double-clicking a tab, you can switch to editing mode and edit the text associated with the tab. The size of the tab is adjusted to fit the text dynamically during your input.

Structure and Design

The ***GridTabControl*** is derived from the ***System.Windows.Forms.Panel*** class. ***GridTabControl*** is a composite control. It contains the object of the ***GridTabBeam*** class, which provides the functionality of a sheet-tab control and provides controls that make the tabs.

NOTE >> All the containing controls, including the ***GridTabBeam***, are accessible through the ***Controls*** collection of the ***GridTabControl***, but do not access the ***GridTabBeam*** directly. Instead, access its functionality through the wrapper functions of the ***GridTabControl***.

The ***GridTabControl*** implements an IExtenderProvider interface and provides two extending properties for all its children controls (except the tab beam):

- `TabLabel`—the string property that represents text associated with the specified control. This text is displayed by a tab associated with a control.
- `TabIdx`—the integer property that represents tab's index for the specified Control. All the tabs appear in order of their TabIdx values.

Other properties that determine the ***GridTabControl*** behavior are:

- `SelectedSheet`—a control type property. It gets or sets the currently selected tab.
- `SelectedIndex`—an integer property. It gets or sets the index of the currently selected tab.

These two properties are synchronized. If `SelectedIndex` property is changed to a new value, the `SelectedSheet` property returns a reference to a corresponding tab ***Control***-derived object.

If any of these properties change the selected tab, the ***SelectedIndexChanged*** event occurs.

The ***GridTabBeam*** control is also a composite control, but it contains windowless objects: four navigation buttons, which are always present on the tab beam, and a variable number of tab buttons.

Most of the functionality of the ***GridTabControl*** is concentrated in the ***GridTabBeam*** class. It holds the information about tab indexes and labels, changes the active tab by reacting to the mouse input, and recalculate the size and position of the tabs.

Design Time Support

The ***GridTabControl*** has the ***GridTabControlDesigner*** designer class associated with it. The main reason for introducing this class is to enable the tab switching functionality and responsiveness to the mouse input at design time. Default designer handles all the mouse input, so the control does not have a chance to react to mouse events.

The ***GridTabControlDesigner***, derived from ***ScrollableControlDesigner***, translates the Windows `WM_MOUSEMOVE` and `WM_LBUTTONDOWN` messages into calls of special design time support functions `DesignerMouseMove` and `DesignerMouseDown` of the designed ***GridTabControl*** control associated with the designer.

Normally an application developer does not need to do anything specific regarding the designer class. This class is used by the Visual Studio.If youderive your own class from the ***GridTabControl*** and want to modify the design time behavior, you might need to derive a new designer class and associate it with the control class.

Using GridTabControl

Below is the list of notes that might be useful for developers who are creating an applications interface with the ***GridTabControl***:

- When first added to the form or containing control, the ***GridTabControl*** is empty and does not contain tabs.
- You can drop the first child control anywhere on the ***GridTabControl***'s surface. It becomes first tab. To add more tabs, drop all subsequent controls on the tab beam area of the ***GridTabControl***. This limitation is necessary in order to provide the ability to add the child controls to the tab-associated controls.
- At runtime, add or remove the child controls of the ***GridTabControl*** by using `AddTab` and `RemoveTab` functions.
- At design time, a mouse click on the surface of any tab-associated child control selects this control and displays its properties in the Properties window. To access the properties of ***GridTabControl*** itself, click the tab beam area of the control.
- At design time, you can select the active tab either by clicking the tab or by changing the SelectedIndex property. The last selected tab at design-time is the first tab activated at runtime, unless you select another tab programmatically.

Chapter 7

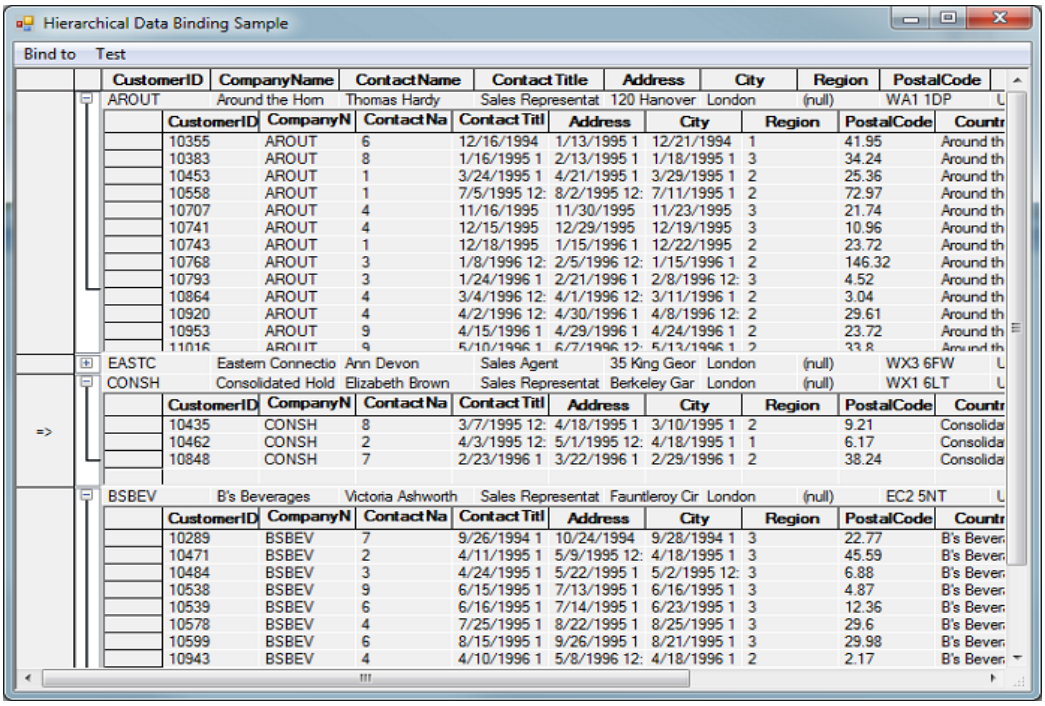
Hierarchical Grids

Introduction

Support for hierarchical grids is included in the Objective Grid for Microsoft .NET ***GridControl*** class. You can establish a one-to-many parent-child relationship between grids. This architecture enables you to expand and collapse each parent row, so that you can view or hide the row's associated children in a tree-like structure. You enable hierarchical grid by setting ***GridControl***'s `HierarchicalGrid` property to `true`.

Figure 29 shows a grid control with hierarchical relationships. When hierarchical display is enabled, each row can serve as parent to a child grid that is a separate instance of ***GridControl*** or a ***GridControl***-derived class. From the parent ***GridControl***, you can retrieve any child ***GridControl*** and access its API directly. Moreover, grids can be nested to any depth.

Figure 29 – Hierarchical Grid Showing Customers and Their Orders



Hierarchical grid features include:

- Grid parents and children that can be instances of different types.
- Child grids that can be instances of different types.
- Virtual child grid initialization that allows custom initialization for each child grid.
- A plus-minus expander that can be selectively shown on a per row basis.
- Seamless current cell movement between parent and child.
- ADO.NET data binding that allows navigation of hierarchical relationships within data sources.

Hierarchical Grid Methods and Properties

When you enable hierarchical grid, the following hierarchical-grid-specific properties and methods are available:

Table 28 – Hierarchical Grid Properties

Property	Description
`HierarchicalGrid`	Enables or disables hierarchical grid display. This property should be set to `true` to enable hierarchy display or `false` to prevent it. Note: This property can be assigned at design time or runtime. If the value is changed at runtime, all rows and columns are automatically deleted and must be programmatically added back to the grid.
`HgMap`	For advanced uses. It is described in "Advanced Topics."

Table 29 – Hierarchical Grid Methods

Method	Description
`EnableHierarchicalGrid`	Enables or disables hierarchical grid display. This method is virtual, allowing your derived classes to override this method and do custom initialization steps before or after the mode is changed.
`GetChildAt`	Retrieves the ***GridControl***-derived instances at the specified row. Once retrieved, you can directly access the API of the child grid to populate it with data or affect its styles.
`GetParent`	From a child grid, this method returns the parent grid or null if this grid is the topmost grid.
`ExpandRow`	Expands or collapses the specified row, showing or hiding the child grid. This method can be overridden, enabling derived classes to perform application-specific actions following a row expansion or collapse.
`IsChildShown`	Returns `true` if the child at the specified position is shown or `false` if hidden.
`HideExpanderColumn`	Enables you to show or hide the expander column. The expander column is the left-most column containing the plus-minus expander controls for all parent rows.
`DeleteChildAt`	Destroys the child associated with the given row, assuming it has been created. Note: Child grid instances are created on demand.
`DeleteChildren`	Destroys all children within the range of rows specified.

Table 29 – Hierarchical Grid Methods (Continued)

Method	Description
`CreateChildAt`	Virtual method that is responsible for creating and initializing a child grid instance for the specified row. This method can be overridden to enable your own ***GridControl***-derived classes to be used as child grid types.
`InstantiateChildAt`	Instantiates the grid control instance to use as the child for the specified row. `CreateChildAt` calls this function, so you can override either method to customize the child grid type.
`InitializeChildAt`	Initializes the child grid just before displaying it.
`IsRowExpandable`	Indicates whether or not the given row contains children and should therefore display a plus-minus expander control.

NOTE >> The non-hierarchical ***GridControl*** API members do not change when hierarchical grid is enabled.

Using Hierarchical Grids

The hierarchical grid interface includes virtual functions that allow you to specify child grid types and custom initialization steps.

To specify a custom grid type for the child grid, you must first create a derived parent grid type so that the virtual methods responsible for creating a child can be overridden.

NOTE >> The steps for creating a .NET control-derived class while retaining designer functionality are not straightforward. "Deriving from GridControl with Designer Functionality" describes the pitfalls. You can find this article in Readme.rtf, which is located in the Docs subdirectory of your Objective Grid for Microsoft .NET installation directory.

1. Following the steps described in "Deriving from GridControl with Designer Functionality," create a new hierarchical grid class with the following declaration:

```
using Stingray.Grid;
…
public class HierGrid : Stingray.Grid.GridControl
{
...
```

2. Replace the ***GridControl*** type that is instantiated in your form class so that this new hierarchical grid type is used instead.

3. Add the following override in the ***HierGrid*** class:

```
public override GridControl InstantiateChildAt(int nRow)
{
  return new ChildGrid();
}
```

   This override modifies the default child grid type that is created when the user expands a row.

4. Define the ChildGrid type by adding the following class declaration after the HierGrid class but in the same file:

```
public class ChildGrid : Stingray.Grid.GridControl
{
...
```

   At this point, the parent and child grid classes are declared and the form is modified to instantiate these types instead.

5. To add scroll bars to the child grids, add the following override to the HierGrid parent class:

```
public override void InitializeChildAt(int nRow,
       GridControl child)
{
  child.SetScrollBarMode(ScrollBarId.Both,
    ScrollBarVisibility.Automatic, false);
  base.InitializeChildAt( nRow, child );
}
```

6. To initialize the parent and child grids with text, add a handler for the ***GridInitialized*** event and add the following code:

```
private void grid1_GridInitialized(object sender,
 System.EventArgs e)
{
  grid1[2,2].Style.Value = "Cell (2,2)";
  gridControl[3,3].Style.Value = "Cell (3,3)";
  Stingray.Grid.GridControl child =
    grid1.GetChildAt(2, true);
  child[2,1].Style.Value = "Child cell (2,1)";
}
```

Advanced Topics

Internally, a hierarchical grid is implemented on top of a non-hierarchical grid, using covered cells and hidden rows. This is an efficient implementation approach, but it has a side effect: rows and columns are not numbered sequentially as they are when hierarchical mode is disabled. For example, when in hierarchical mode, the second logical row on your screen might actually be the fifth physical row with several hidden rows preceding it.

Logical cell coordinates must be transformed into physical cell coordinates before the grid can actually be accessed or altered. ***GridControl*** does the transformation for you. All API members expect logical cell coordinates and transform the arguments into physical cell coordinates. Additionally, overridden methods receive parameters already transformed into logical coordinates.

You will need to use physical coordinates directly or transform between logical and physical coordinate systems in two cases:

- You want to call a method, passing the expander column or a hidden row.
- You want to override a method and take action when a hidden row is passed in.

To specify physical coordinates, use the `HgMap` property.

The HgMap Property

The ***GridControl***'s `HgMap` property returns an interface to the Hierarchical Grid Mapper object, which manages the automatic transformation between logical and physical coordinates. Given the `HgMap` object, you can temporarily or permanently suspend the automatic transformations, and do the transformations manually instead. In addition, when executing overridden methods, the `HgMap` object gives you access to the original physical row and column arguments that apply to the current call context.

The object returned by the `HgMap` property is an instance of the ***HierGridMap*** class. The methods defined on this class follow:

Table 30 – Methods of the HierGridMap Class

Method	Description
`Suspend`	Temporarily suspends automatic mapping of logical cell coordinates to physical.
`Resume`	Following a `Suspend`, this method resumes the logical to physical mapping.
`Enable`	Enables or disables the logical to physical transformations and cancels storage of current call physical row and column parameters.
`LtoP.RowLtoP. ColLtoP.Range`	Converts a row, column, or range from logical to physical units.
`PtoL.RowPtoL. ColPtoL.Range`	Converts a row, column, or range from physical to logical units.

Table 30 – Methods of the HierGridMap Class (Continued)

Method	Description
`RowRow2Row3`	Retrieves the first, second, or third row parameter for the current call context. Typically, these methods are used from overridden methods to access the physical row parameters original passed.
`ColCol2Col3`	Retrieves the first, second, or third column parameter for the current call context.
`Range`	Retrieves the range parameter for the current call context.

Using the HgMap Property

The expander column is skipped in column numbering, so it is not programmatically possible to expand the expander column without using the `HgMap` property. The following code resizes the expander column to 25 pixels:

```
HgMap.Suspend();
Grid.SetColWidth(1, 1, 25);
HgMap.Resume();
```

The `Suspend` and `Resume` functions prevent automatic logical to physical coordinate mapping just for the `SetColWidth` call. If you need to override a function and specify a style for the expander column, you can do it as follows:

```
public override bool OnGetStyleRowCol(GetStyleRowColEventArgs e)
{
  if (HierarchicalGrid)
  {
    if (HgMap.Col == 1) // Expander column is col 1
    {
      // Set e.Style here
      …
```

Chapter 8

1stGrid Tutorial

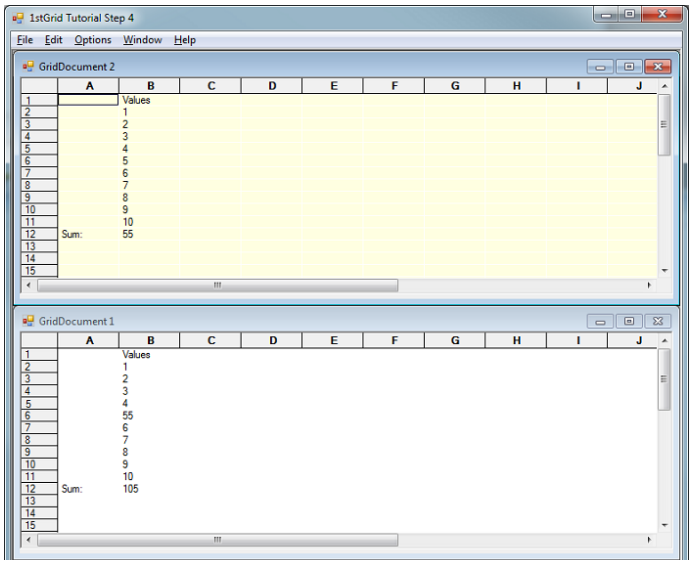
Introduction

This tutorial shows how to use Objective Grid for Microsoft .NET to create a simple grid application, 1stGrid, and the product's custom control is used in a number of ways. A Microsoft Solution file with complete source code for this tutorial's projects is included with the distribution.

This tutorial is organized into four major projects, Steps 1-4 described below.

- **Step 1**—A simple Windows Forms application is created. The application contains a single, main form containing a Objective Grid for Microsoft .NET grid control.
- **Step 2**—An MDI grid application is created. The application contains a menu system, and allows for a number of grid editing functions.
- **Step 3**—A property grid is added to the grid application from Step 2. This shows how to modify the properties for a grid control at run time.
- **Step 4**—Formula Engine support is added to the grid application from Step 3. Grid events are handled to initialize the grid and to validate cell data.

Figure 30 – Completed 1stGrid Application



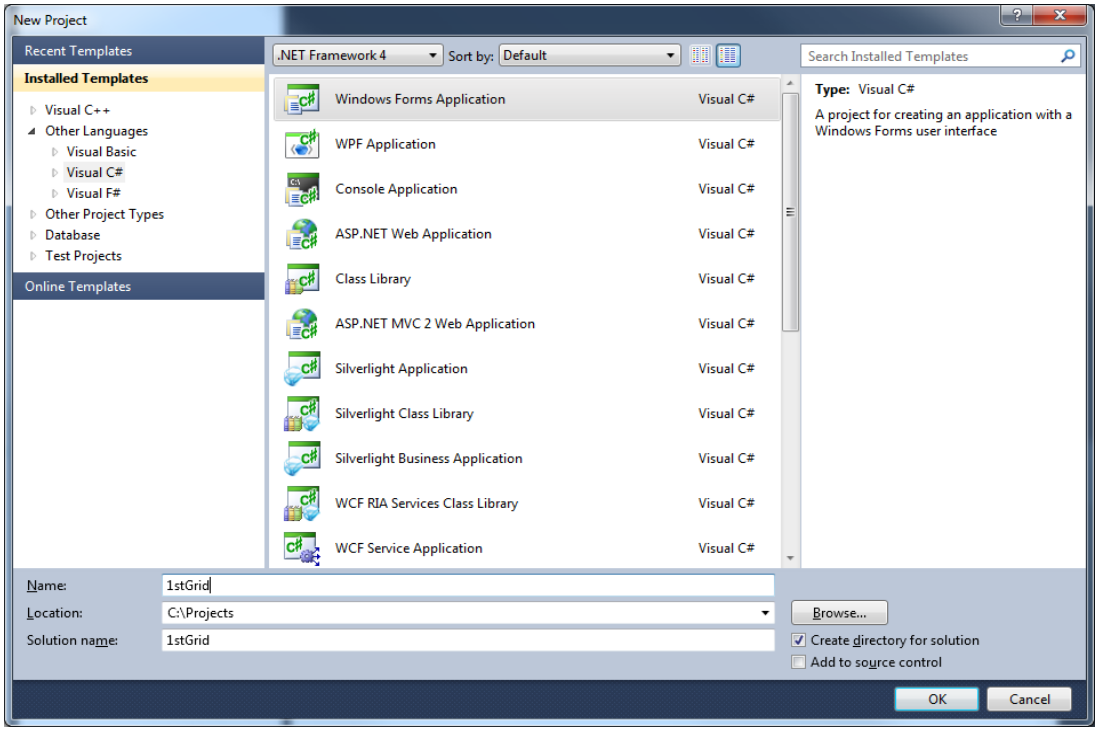
1stGrid - Step 1

This section lists the steps required to create the first iteration of the 1stGrid application. The first iteration of the application is a very simple Windows Forms application with a single form containing one Objective Grid for Microsoft .NET grid control. The first two cells in column one of the grid are initialized with the text “Hello” and “World”, respectively.

Create a New Solution and Project

1. Open Microsoft Visual Studio.
2. Click **Visual C# Projects** to launch the New Project dialog (Figure 31).
3. Select **Windows Forms Application**.

Figure 31 – New Project Dialog



4. Name the project "1st Grid" and click **OK**. A main form (Form1) is automatically added to the project and displayed in design mode (Figure 32). Resize the form as desired.
5. Select Form1. If the properties for Form1 are not already displayed, then right click the form to display a context menu. Select **Properties** from the context menu.
6. In the property grid for the form, change the following property:

   Set `Text` to "1stGrid Tutorial Step 1".

Add a GridControl

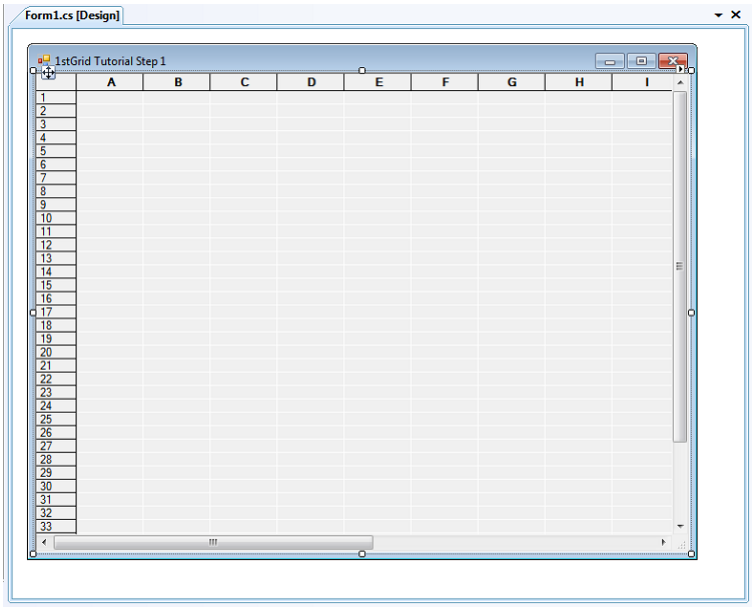
1. You are now ready to add a Grid Control to Form1. Access the Windows Forms Toolbox, by selecting the **View | Toolbox** menu option.
2. Select and drag a GridControl from the Toolbox onto the form.
3. Size and position the grid as desired.
4. Select the Grid Control. In the property grid for the control, set the properties shown in Table 31

Table 31 – GridControl Properties

Property	Value
`DrawGrid`	`false`
`BorderStyle`	Sunken
`Anchor`	Top, Bottom, Left, Right
`AutoScroll`	true
`ColCount`	15
`RowCount`	40

NOTE >> `DrawGrid` is a property of the form; it is not a Rogue Wave property.

Figure 32 – 1stGrid Step 1 Form

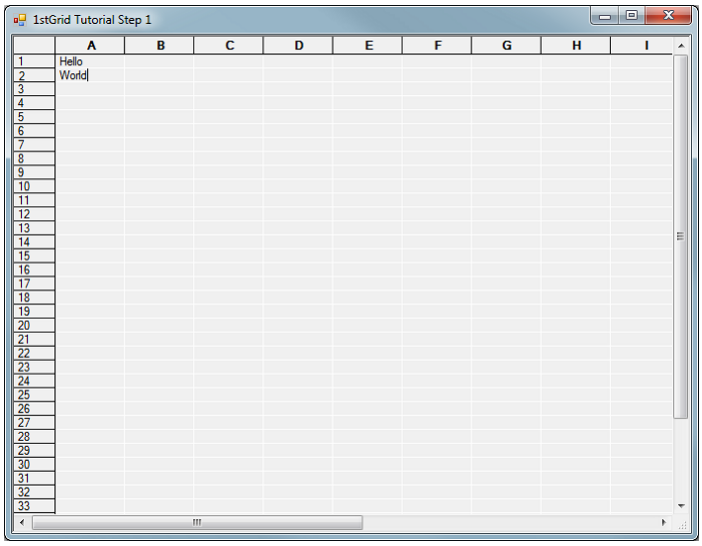


5. Select the Grid Control in the form.
6. Click the events button in the properties grid (the lightning bolt).
7. Double click the `GridInitialized` event (in Grid Control Events). Add the following code to handle the event.

```
gridControl1[1,1].Style.Value = "Hello";
gridControl1[2,1].Style.Value = "World";
```

8. Build and run the application.

Figure 33 – Completed 1stGrid Step 1



1stGrid - Step 2

In Step 2, you will change the Step 1 application in several ways:

- You will make it an MDI application.
- You will add a menu system with commands for a number of common grid operations
- You will add an "About" dialog.

This section requires the most work of any of the steps in this tutorial.

Create a New Project

First, add a new project to the 1stGrid solution for the Step 2 application.

1. In the Solution Explorer, right click on the 1stGrid solution.
2. In the resulting context menu, select the **Add | New Project** option.
   - Click **Visual C# Projects**.
   - Select **Windows Application**.
   - Set the name to "Step 2".
   - Click **OK**.
3. Right-click Step 2.
   - In the context menu, select **Set As Startup Project**.
4. Resize the form as desired.
5. In the property grid for the form, change the following properties:
   - Set `Text` to `1stGrid Tutorial Step 2`.
   - Set `IsMdiContainer` to `true`.

Add a Menu

1. If the Toolbox is not already visible, then make it visible using the **View | Toolbox** menu option.
2. Drag a main menu from the Toolbox onto the form.
3. Click `mainMenu1` at the base of the form designer.
4. You will now add all of the menu options for the application.
   - Click on the menu text "Type Here".

— Type “&File”.

— A drop down menu appears in the designer (to be designed). Add the menu options in Table 32 to the File menu. To add a separator, right-click the Menu Designer, and then click Insert Separator.

Table 32 – File Menu Items

&New
&Close
separator
E&xit

— Click the main menu item to the right of file, to add the Edit menu.

— Add the menu options in Table 33 to the Edit menu. To add a separator, right-click the Menu Designer, and then click Insert Separator.

— Type “&Edit”.

Table 33 – Edit Menu Items

&Undo
&Redo
separator
&Copy
Cu&t
&Paste
C&lear
separator
&Find
R&eplace

— Click the main menu item to the right of file, to add the Window menu.

— Type “&Window”

— Add the following items to the Window menu.

Table 34 – Windows Menu Items

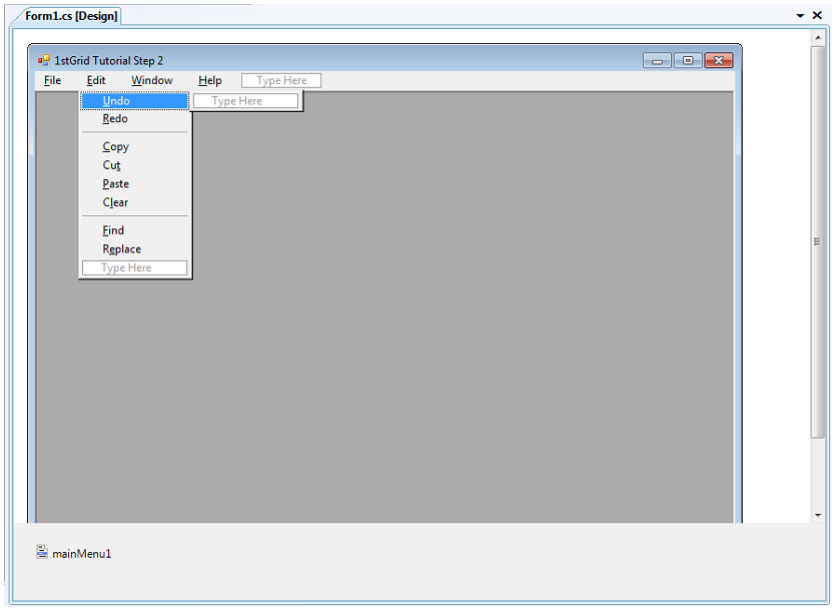
&Cascade
Tile &Horizontal
Tile &Vertical

— Click the main menu item to the right of file, to add the Help menu.

— Type "&Help"

— Add the item "&About" to the Help menu.

Figure 34 – Menu Design



5. Now, for each menu item, do the following. First, select the item. In the property grid, rename the item, giving it a more meaningful name. Then give it a shortcut (if desired). Suggested names and shortcuts follow.

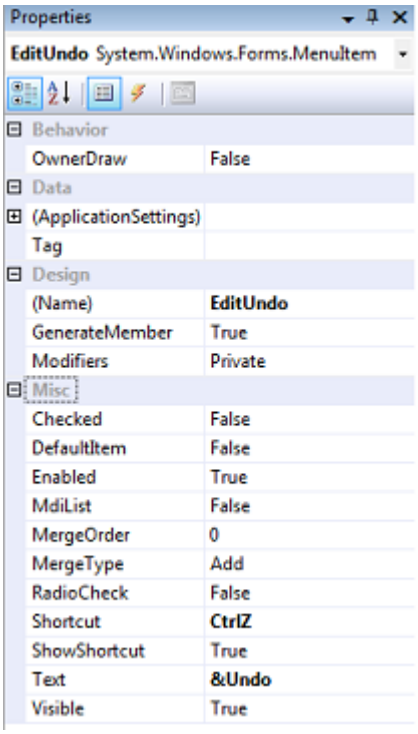
Table 35 – Suggested Menu Item Names and Shortcuts

FileMenu
FileNew (CtrlN)
FileClose
FileExit
EditMenu
EditUndo (CtrlZ)

Table 35 – Suggested Menu Item Names and Shortcuts

EditRedo (CtrlR)
EditCopy (CtrlC)
EditCut (CtrlX)
EditPaste (CtrlV)
EditClear
EditFind (CtrlF)
EditReplace (CtrlH)
WindowMenu
WindowCascade
WindowTileHorizontal
WindowTileVertical
HelpMenu
HelpAbout

Figure 35 – Menu Command Design



Add the GridDocument Class

The next steps add a ***GridDocument*** class to the application. A ***GridDocument*** is a document class that represents a document. Each ***GridDocument*** contains a grid that can be manipulated by the user.

1. From the Solution Explorer, right click Step2 (context menu).
2. Select **Add | Add Windows Form**.
   - Select **Windows Form**.
   - Set the name to `GridDocument.cs`.
   - Click **Open**.
3. Size the form as desired.
4. Select the ***GridDocument*** form.
5. Make the following changes to the form's properties.
   - Set `Text` to `Grid Document`.
6. Select and drag a ***GridControl*** from the Toolbox onto the form.
7. Size and position the grid in the form, as desired.
8. Select the grid control. In the property grid for the control, set the properties shown in Table 36

Table 36 – GridControl Properties

Property	Value
`DrawGrid`	`false`
`BorderStyle`	Sunken
`Anchor`	Top, Bottom, Left, Right
`AutoScroll`	true
`ColCount`	15
`RowCount`	40

NOTE >> `DrawGrid` is a property of the form; it is not a Rogue Wave property.

9. From the **View** menu, select **Code** to edit the source code for the form.
10. Modify the `GridDocument` constructor to accept one argument, a string containing the document name. Also, after the comment, "TODO: Add any constructor code after InitializeComponent call", add the line:

```
this.Text = docName;
```

The code now looks like this:

```
public GridDocument(string docName)
{
   //
   // Required for Windows Form Designer support
   //
   InitializeComponent();

   //
   // TODO: Add any constructor code after InitializeComponent call
   //
   this.Text = docName;
}
```

11. Switch to Class View. If the class view is not visible, you can access it using the **View | Class View** menu option.

12. Expand the class view tree as follows: **Step 2** | **NameSpaces** | **Step_2** |**GridDocument**.

13. With the ***GridDocument*** class visible, right click the ***GridDocument*** class.

    — From the context menu that appears, select **Add | Add Property**. In the C# Property Wizard, fill in the properties as follows:

Table 37 – GridDocument Properties

Property	Value
`access`	public
`type`	GridControl (type this; it is not in the Property type list)
`name`	Grid
`Accessors`	get
`Modifiers`	none
`Comment`	"Get the grid control embedded in this form"

    — Click **Finish**.

    — The source code editor appears. Edit the generated property code as follows.

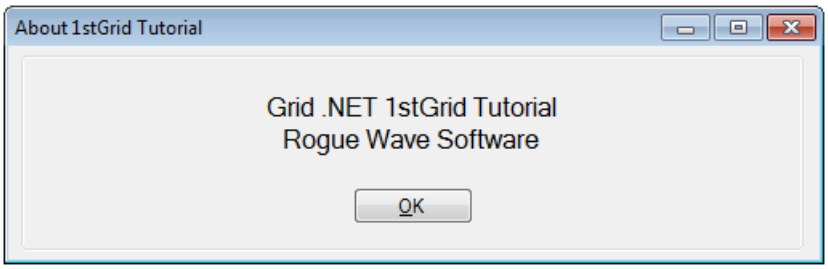
```
/// <summary>
/// Get the grid control embedded in this form
/// </summary>
public Stingray.Grid.GridControl Grid
{
  get
  {
    return gridControl1;
  }
}
```

14. Close the source code window and the design view for GridDocument.

Add an “About” Box

1. Return to the Solution Explorer.
2. Right click on Step 2.
3. From the context menu that appears, select **Add | Add Windows Form**.
   - Select **Windows Form**.
   - Set the name to `HelpAbout.cs`.
   - Click **Open**.
4. Size the form as desired.
5. Select the form. Make the following changes to the form properties:
   - Set `FormBorderStyle` to `FixedDialog`.
   - Set `Text` to “About 1stGrid Tutorial”.
   - Set `StartPosition` to `CenterParent`.
   - Set `SizeGridStyle` to `Hide`.
6. Drag a group box from the Toolbox onto the form.
7. Select the group box.
8. Change the `Text` property for the group box to “” (no text).
9. Size the group box to form a border around the perimeter of the window.

Figure 36 – About Dialog Box



10. Drag a label from the Toolbox onto the form.
11. Select the label, and change its properties as follows:
    - Set `Font` as desired.

- Set `Text` to describe the application.
- Set `TextAlign` to `MiddleCenter`.

12. Size the label as needed, in order to properly display the text.
13. With the label selected, center the label in the form with the **Format | Center in Form | Horizontally** menu option from the Visual Studio main menu.
14. Drag a button from the Toolbox onto the form.
15. With the button selected, center the Button in the form with the **Format | Center in Form | Horizontally** menu option.
16. Ensure the button is still selected. Change its properties as follows:
    - Set `Text` to `&OK`.
    - Set `Name` to `OK`.
17. Click the events button in the property grid (the lightning bolt).
18. Double-click the `Click` event.
19. Add this code for the button's click event:
    ```
    this.Close();
    ```
20. Close the source code editor.
21. Close the design view for `HelpAbout`.

Add MDI Support

You will now update the source code for Form1 to add support for the MDI interface.

1. View the source code for Step 2 `Form1.cs`.
2. Add a new private `int` property, the document counter:
    ```
    private int documentCount;
    ```
3. Add the following code after the "TODO: Add any constructor code after InitializeComponent call" comment:
    ```
    //Add an initial doc
    AddDocument();
    ```
4. Add the following implementation for the `AddDocument` method:
    ```
    // Add a document
    private void AddDocument()
    {
      documentCount++;
      GridDocument doc = new GridDocument("GridDocument " + documentCount);
    ```

```
    doc.MdiParent = this;
    doc.Show();
  }
```

Add Event Handlers

Finally, you will add event handlers for all of the menu options in the application menu.

1. Go to the design view for `Form1.cs`.
2. First, you will implement the **File | New** event handler.
   - Double-click the **File | New** menu option.
   - Add the following code:
     ```
     AddDocument();
     ```
3. Next, implement the **File | Close** event handler.
   - Double-click the **File | Close** menu option.
   - Add the following code to handle the event:
     ```
     GridDocument doc = (GridDocument) this.ActiveMdiChild;
     if (doc != null)
       this.ActiveMdiChild.Close();
     ```
4. The following source code shows the implementations for the remaining menu item click event handlers.
   ```
   private void FileExit_Click(object sender, System.EventArgs e)
   {
     this.Close();
   }

   private void EditUndo_Click(object sender, System.EventArgs e)
   {
     GridDocument doc = (GridDocument) this.ActiveMdiChild;
     if (doc != null)
       doc.Grid.OnUndo();
   }

   private void EditRedo_Click(object sender, System.EventArgs e)
   {
     GridDocument doc = (GridDocument) this.ActiveMdiChild;
     if (doc != null)
       doc.Grid.OnRedo();
   }

   private void EditCopy_Click(object sender, System.EventArgs e)
   {
     GridDocument doc = (GridDocument) this.ActiveMdiChild;
     if (doc != null)
   ```

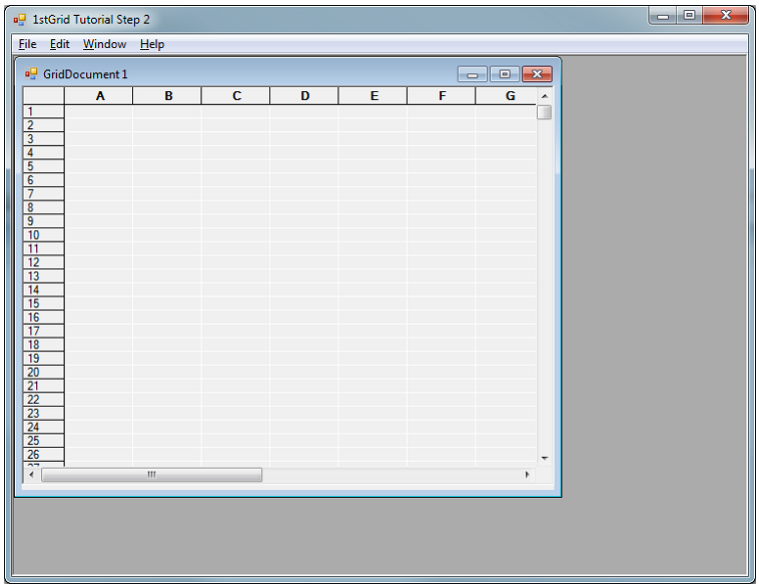
```
    doc.Grid.OnCopy();
}

private void EditCut_Click(object sender, System.EventArgs e)
{
  GridDocument doc = (GridDocument) this.ActiveMdiChild;
  if (doc != null)
    doc.Grid.OnCut();
}

private void EditPaste_Click(object sender, System.EventArgs e)
{
  GridDocument doc = (GridDocument) this.ActiveMdiChild;
  if (doc != null)
    doc.Grid.OnPaste();
}

private void EditClear_Click(object sender, System.EventArgs e)
{
  GridDocument doc = (GridDocument) this.ActiveMdiChild;
  if (doc != null)
    doc.Grid.OnClear();
}

private void EditFind_Click(object sender, System.EventArgs e)
{
  GridDocument doc = (GridDocument) this.ActiveMdiChild;
  if (doc != null)
    doc.Grid.OnFind();
}

private void EditReplace_Click(object sender, System.EventArgs e)
{
  GridDocument doc = (GridDocument) this.ActiveMdiChild;
  if (doc != null)
    doc.Grid.OnReplace();
}

private void WindowCascade_Click(object sender, System.EventArgs e)
{
  this.LayoutMdi(MdiLayout.Cascade);
}

private void WindowTileHorizontal_Click(object sender, System.EventArgs e)
{
  this.LayoutMdi(MdiLayout.TileHorizontal);
}
```

```
private void WindowTileVertical_Click(object sender, System.EventArgs e)
{
  this.LayoutMdi(MdiLayout.TileVertical);
}

private void HelpAbout_Click(object sender, System.EventArgs e)
{
  new HelpAbout().ShowDialog();
}
```

5. Build and run the application.

Congratulations! You've completed the longest portion of the 1stGrid tutorial.

Figure 37 – Completed 1stGrid Step 2



1stGrid - Step 3

This section outlines the steps for step 3 of the 1stGrid tutorial. Step 3 adds a property grid to the application, allowing the properties of a grid control to be configured at run-time.

If you don't want to keep Step 2, then skip to "Add a Property Grid" to continue adding the Step 3 features to Step 2.

Create a New Project

1. Right click `Solution '1stGrid'` in Solution Explorer.
2. From the context menu that appears, select **Add | New Project**.
   - Click **Visual C# Projects**.
   - Select **Windows Application**.
   - Set the `Name` to "Step 3".
   - Click **OK**.
3. Right click and delete `AssemblyInfo.cs` and `Form1.cs`.
4. From the command prompt or from the Windows Explorer:
   - Delete all of the remaining Step 3 project and source files.
   - Copy all of the Step 2 source and project files into the Step 3 directory.
   - Rename `Step 2.*` to `Step 3.*`.
   - Visual Studio displays a message that the project has changed. Select Discard to discard any cached Step 3 project files.
5. Right click **Step 3**.
6. From the context menu that appears, select **Set As Startup Project**.

Add a Property Grid

The following steps create a new form for displaying the property grid.

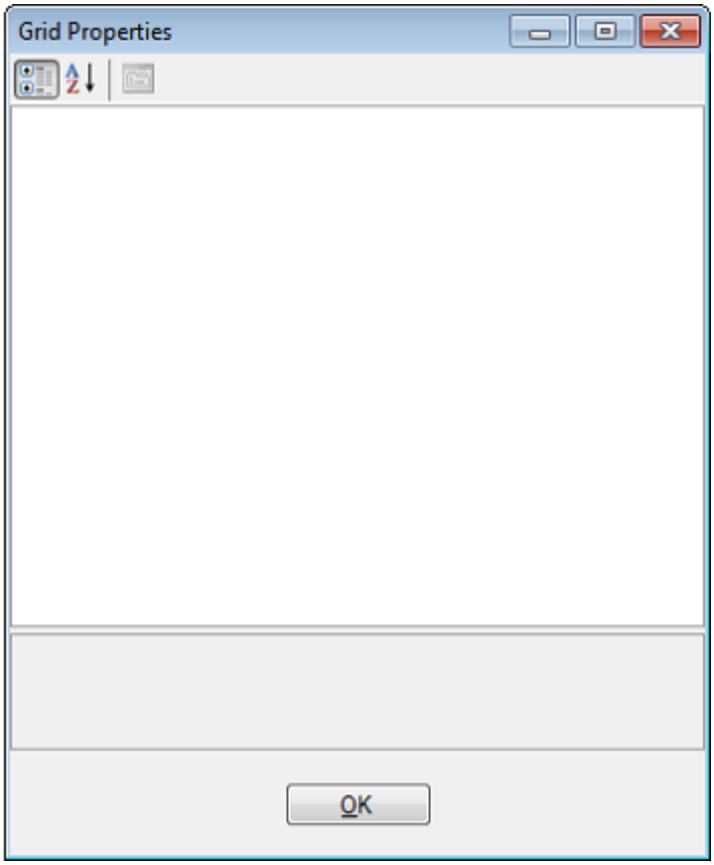
1. Open `Form1.cs` in design view.
2. Select the form.
   - Set the `Text` property for the form to "1stGrid Tutorial Step 3".

3. Add a new form to hold the property grid.
   - Right click **Step 3**.
   - From the context menu that appears, select **Add | Add Windows Form**.
   - Select **Windows Form**.
   - Set the `Name` to `GridProperties.cs`.
4. Size the form as desired.
5. Change the following properties for the form:
   - Set `FormBorderStyle` to `FixedDialog`.
   - Set `Text` to "Grid Properties".
   - Set `StartPosition` to `CenterParent`.
6. Drag a property grid from the Toolbox onto the form. Size as desired.

NOTE >> The Property Grid is not present by default in the Toolbox. To add it, right-click the Toolbox, choose "Customize Toolbox...", select the .NET Framework Components tab, check the box next to Property Grid, and click OK.

7. Drag a button from the Toolbox onto the form. Position as desired.
8. Set the properties for the button.
   - Select the button.
   - View the properties for the button.
   - Set `Text` to `&OK`.
   - Set `Name` to `OK`.

Figure 38 – Grid Properties Form



9. Add the implementation for the `Click` event.
   - View the events for the button by clicking the lightning bolt in the property grid.
   - Double-click the `Click` event.
   - Add the following implementation for the event:

```
this.Close();
```

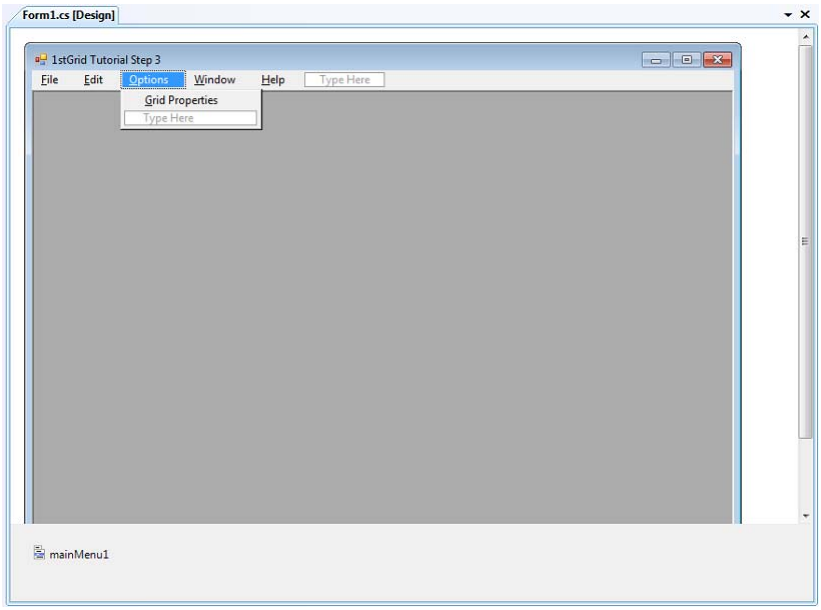
10. View the source code for the form (use the **View|Code** Visual Studio menu option).
11. Add a constructor argument of type `Stingray.Grid.GridControl`.
12. After the comment "TODO: Add any constructor code after InitializeComponent call", add a command to set the selected object for the property grid to the grid given to the constructor.

```
public GridProperties(Stingray.Grid.GridControl grid)
{
//
// Required for Windows Form Designer support
//
InitializeComponent();
```

```
//
// TODO: Add any constructor code after InitializeComponent call
//
propertyGrid1.SelectedObject = grid;
}
```

13. Open the design view for `Form1.cs`.
14. Right click the Window menu and select **Insert New** from the context menu that appears.
15. Add the menu &Options. Set the `Name` property for the menu to `OptionsMenu`.
16. Add the item &Grid Properties to the Options menu. Set the `Name` property of this item to `OptionsGridProperties`.

Figure 39 – Adding the Options Menu

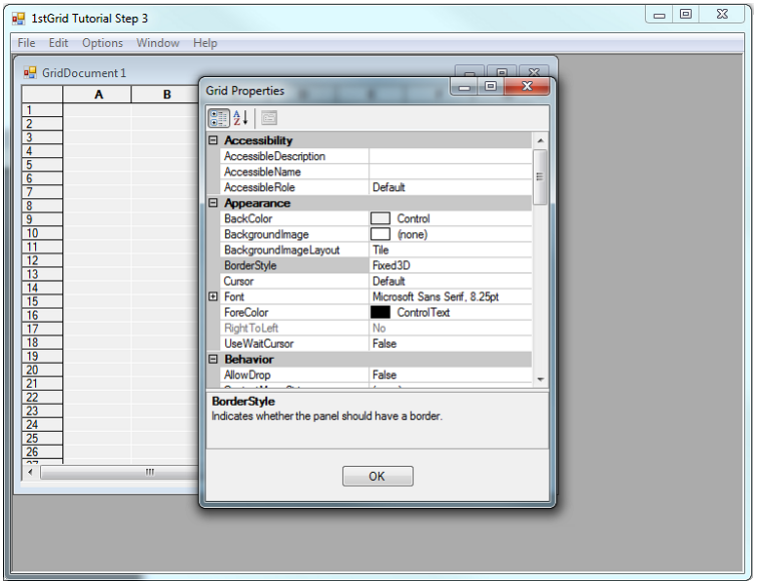


17. Double-click the GridProperties menu item, and add the following code for the implementation of the event handler.

```
GridDocument doc = (GridDocument) this.ActiveMdiChild;
if (doc != null)
  new GridProperties(doc.Grid).ShowDialog();
```

18. Build and run the application.

Figure 40 – Completed 1stGrid Step 3



1stGrid - Step 4

This section gives the steps required to complete Step 4 of the 1stGrid tutorial. Step 4 enables the Objective Grid for Microsoft .NET formula engine, and configures the grid documents to sum numbers in column two, from row 2 through row 11.

If you don't want to keep Step 3, then skip to "Enable the Formula Engine" to continue adding the Step 4 features to Step 3.

Create a New Project

1. Right click `Solution '1stGrid'` in Solution Explorer.
2. From the context menu that appears, select **Add | New Project**.
   - Click **Visual C# Projects**.
   - Select **Windows Application**.
   - Set `Name` to "Step 4".
   - Click **OK**.
3. Right click and delete `AssemblyInfo.cs` and `Form1.cs`.
4. From the command prompt or from the Windows Explorer:
   - Delete all of the remaining Step 4 project and source files.
   - Copy all of the Step 3 source and project files into the Step 4 directory.
   - Rename `Step 3.*` to `Step 4.*`.
5. Visual Studio will indicate that the project has changed. Select **Discard** to discard any cached Step 4 project files.
6. Right click Step 4.
7. From the context menu that appears, select **Set As Startup Project**.

Enable the Formula Engine

The following steps enable the formula engine.

1. Open `Form1.cs` in design mode.
2. Select the form. Change the following form properties for Form1:
   - Set `Text` to "1stGrid Tutorial Step 4".

3. Open the Design View for the ***GridDocument*** class.
4. Select the grid control in the ***GridDocument*** form.
5. Select the properties for the Grid Control.
6. Expand the `Features` property (in the Configurations category).
   - Set `EnableFormulaEngine` to `true`.
   - Set `EnableWorkSheetFunctions` to `true`.

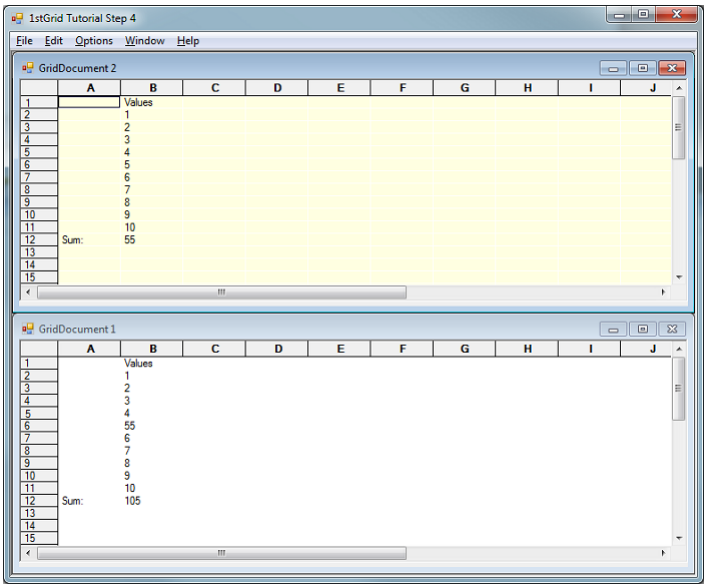
Add Event Handlers

The following steps add event-handling code for the `GridInitialized` and `EndEditing` events. You handle the `GridInitialize` event by adding some simple titles and a formula to the grid. The formula sums the numbers in column 2, rows 2 through 11. You handle the `EndEditing` event by validating any user input, which should be empty text or numeric data.

1. Select the events for the Grid Control in the property grid (click on the lightning bolt in the property grid).
2. Double click the `GridInitialized` method, and add the following implementation for the `GridInitialized` method:
```
gridControl1[1,2].Style.Value = "Values";
gridControl1[12,1].Style.Value = "Sum:";
gridControl1[12,2].Formula = "=SUM(B2:B11)";
```
3. Select the events for the Grid Control in the property grid (click the lightning bolt).
4. Double click the `EndEditing` event, and add the following validation code:
```
Cell cell = gridControl1.CurrentCell;
if (cell.Col == 2 && cell.Row >= 2 && cell.Row <= 11)
{
  bool isDigits = true;
  for (int i = 0; isDigits && i < cell.Style.Value.Length; ++i)
  {
    isDigits = Char.IsDigit(cell.Style.Value, i);
  }

  if (!isDigits)
  {
    MessageBox.Show("Invalid Numeric Value");
    e.IsValid = false;
  }
}
```
5. Add the following using declaration at the top of the file `GridDocument.cs`:
```
using Stingray.Grid;
```
6. Build and run the completed application.

Congratulations! You've completed the final step of the 1stGrid tutorial.

Figure 41 – Completed 1stGrid Step 4



Chapter 9

Samples

Introduction

Samples are located in the Samples subdirectory of your Objective Grid for Microsoft .NET installation directory. They include:

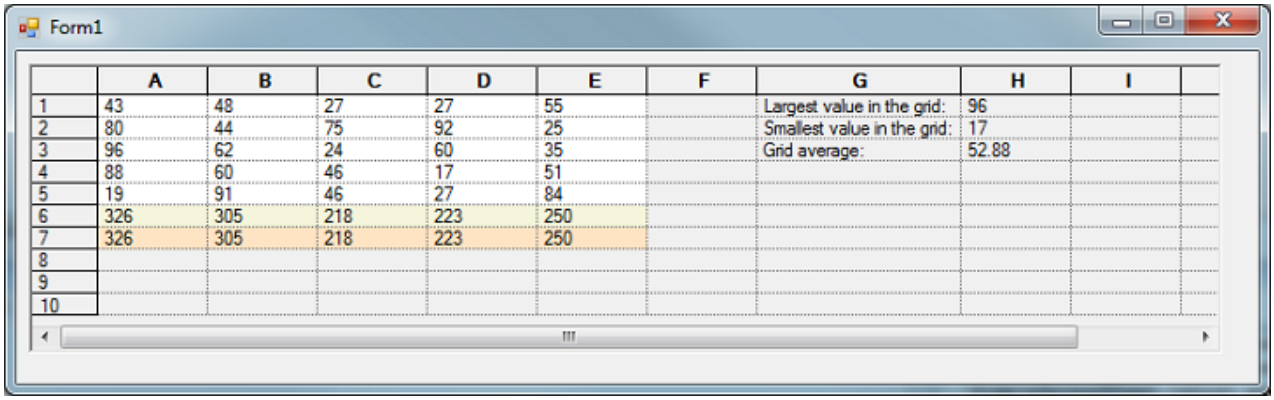
- SimpleSpreadSheet—Shows basic spreadsheet functionality.
- ExcelLike—Demonstrates Objective Grid for Microsoft .NET features that mimic some of the behaviors of Microsoft Excel.
- Virtual Grid—Shows how to store and retrieve grid data from a data structure.
- RaceAttendance—Demonstrates properties of the Stingray.Grid.Style class, and demonstrates printing, print preview, saving grid data to an HTML file, event handling, and embedding a properties sheet so that the grid and cell properties can be manipulated at runtime.
- Simple .NET Control—Shows the most basic use-case for embedded .NET controls: using a .NET `NumericUpDown` control as a cell editor.
- Multiple .NET Controls—Shows how to use several different .NET controls as cell editors, and shows how to make use of CellControl properties and events to customize the behavior of the .NET controls.
- Custom .NET Control—Shows how to create a custom .NET control consisting of a `TextBox` with a button on the right of the text, and shows how to use the custom .NET control as a grid cell editor.
- EventHandlingByDelegation—Shows how to handle grid events using delegates.

- EventHandlingByInheritance—Shows how to handle grid events by deriving from GridControl and overriding virtual event handlers.
- Print—Shows how to utilize the print and print preview functionality in the grid.

The SimpleSpreadSheet, ExcelLike, Virtual Grid, and RaceAttendance samples are described in the following sections.

SimpleSpreadSheet

The SimpleSpreadSheet is a basic example of spreadsheet functionality in Objective Grid for Microsoft .NET. This sample demonstrates simple formatting; see "RaceAttendance" for a more comprehensive example.



The sample consists of a series of randomly generated numbers. Sums for each column are at the bottom of each column. The ranges are specified in two ways:

- In row 6, the cells in the summation is explicitly specified: **A1+A2+A3+A4+A5**.
- In row 7, the cells in the summation is specified as a range: **A1..A5**.

For more information on functions, see Section 2.5, "Formula Engine Support."

ExcelLike

The ExcelLike sample demonstrates Objective Grid for Microsoft .NET features that mimic features of Microsoft Excel:

- Current cell
- Headers
- Scrolling
- Selection frame

This sample also demonstrates:

- Find and replace
- Formula engine and worksheet functions
- Scroll tips
- Cell and tool tips
- Saving a grid in HTML format

The ExcelLike sample is a multiple document interface type of application with a frame containing zero or more grid child windows.

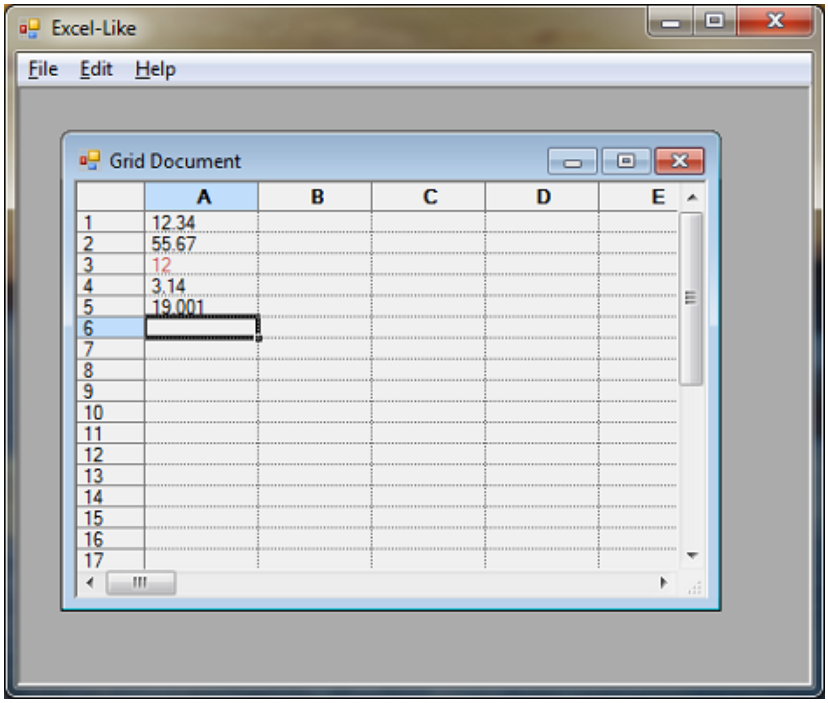
Running the Sample

After building the sample, you can run it by double-clicking `ExcelLike.exe` in the `bin\debug` or `bin\release` directories under the `Samples\ExcelLike` directory. The main frame window of the application is displayed.

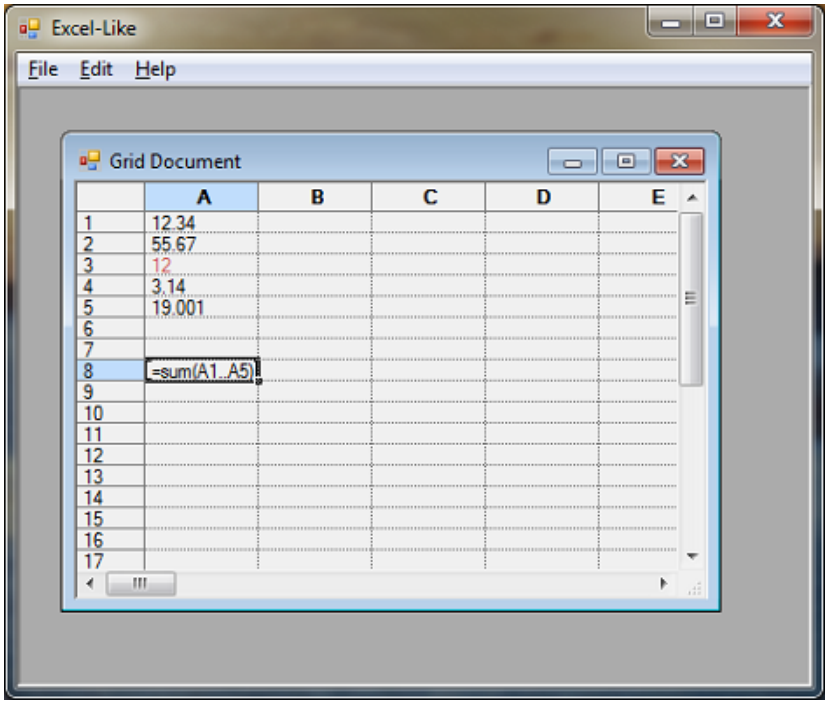
Press **Ctrl-N** or select **File | New** to create and display a new grid. You may have any number of grids open at one time, within your system's memory limits. For this part of the example, you need just one grid.

Formula Support

1. In columns A1 through A5, enter some integer and floating point numbers:



2. Add those numbers using the formula `=sum(A1..A5)` in cell A8:

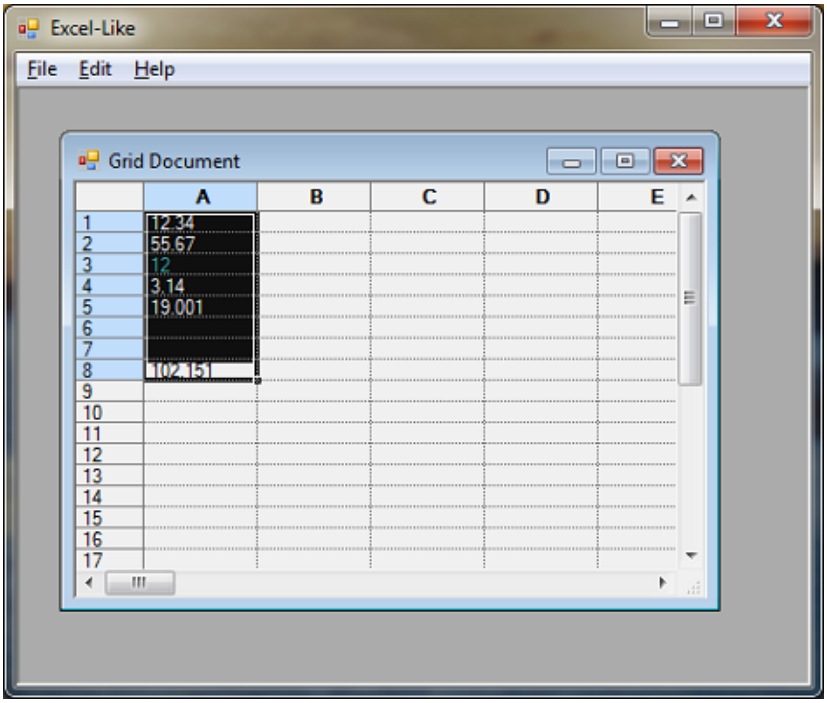


3. Press enter, and you should see the formula replaced with the result of the formula. If you reselect the cell, the formula is displayed.

You can get a similar result by explicitly referencing each cell in the summation range: `=sum(A1+A2+A3+A4+A5)`.

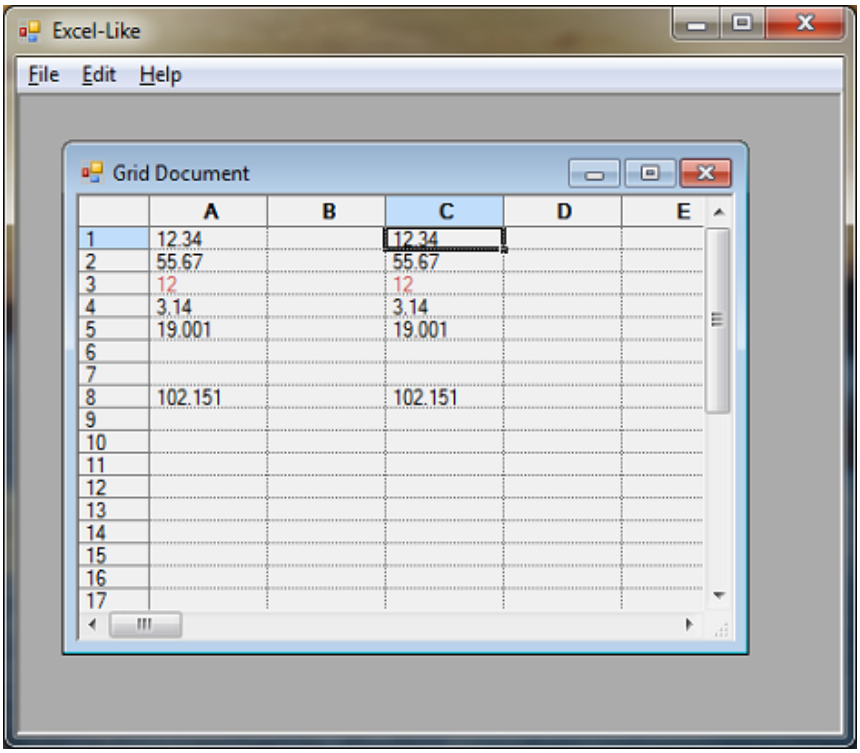
Copying and Pasting Data and Formulas

1. Select the cells in the range A1 through A8: Click on A1, and then hold down the left mouse button while dragging the selection rectangle down to cell A8.

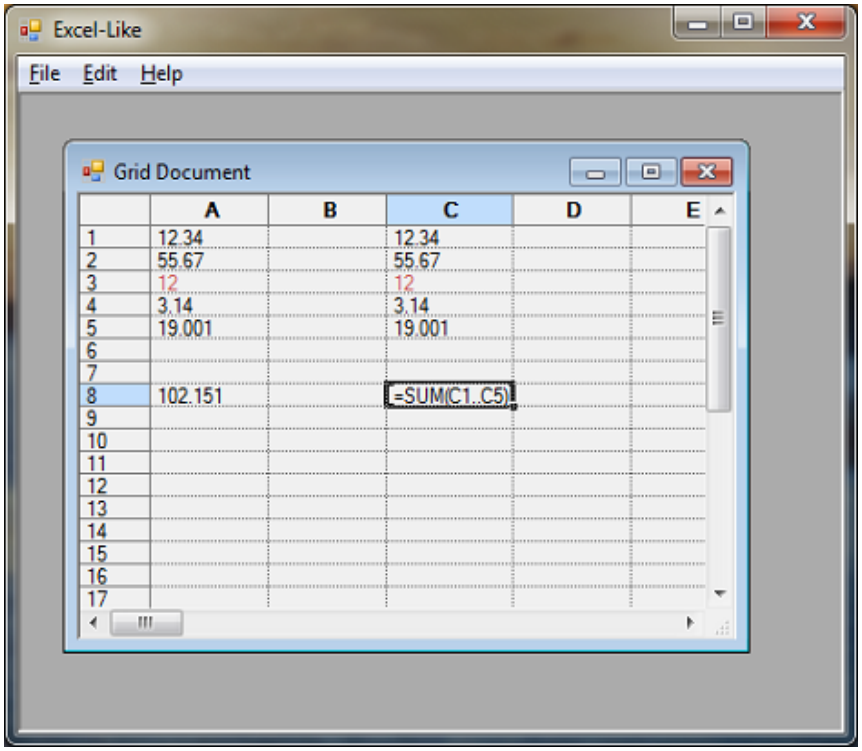


2. Select **Edit | Copy** from the menu bar to copy the selected range of cells to the clipboard.

3. Select cell C1, and then select **Edit | Paste**.



4. Select cell C8 to reveal the formula. The cell range in the sum changed from A1..A5 to C1..C5.



5. To delete a range of selected cells, highlight the cells, and then select **Edit | Delete**.

Undo and Redo Changes

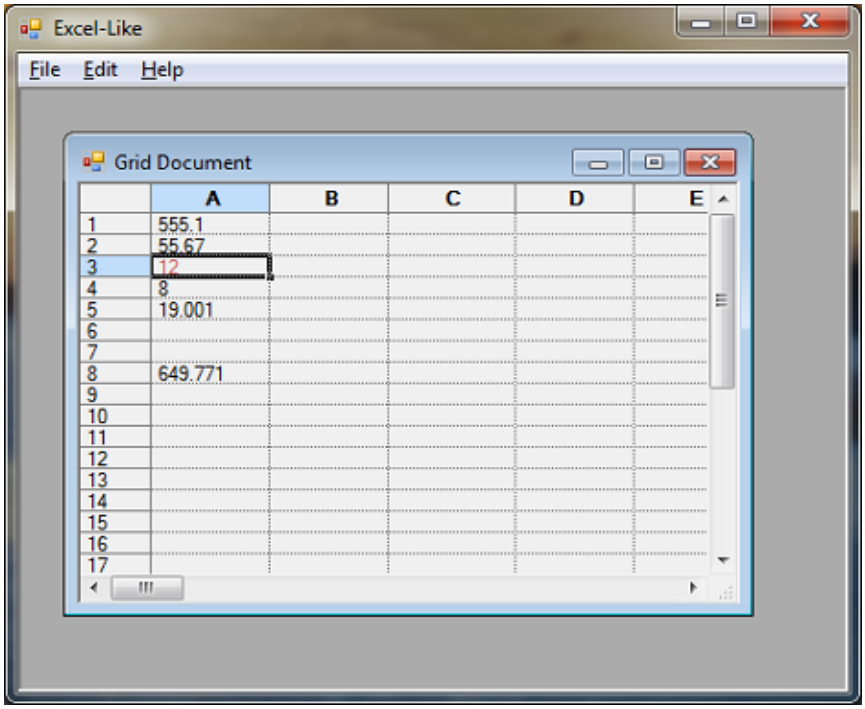
1. Change some values in the range A1..A5, such as 555.1 in A1 and 8 in A4.
2. Select **Edit | Undo** and **Edit | Redo** a few times.

Changes in value and movement of the active cell selection box are affected by Undo and Redo.

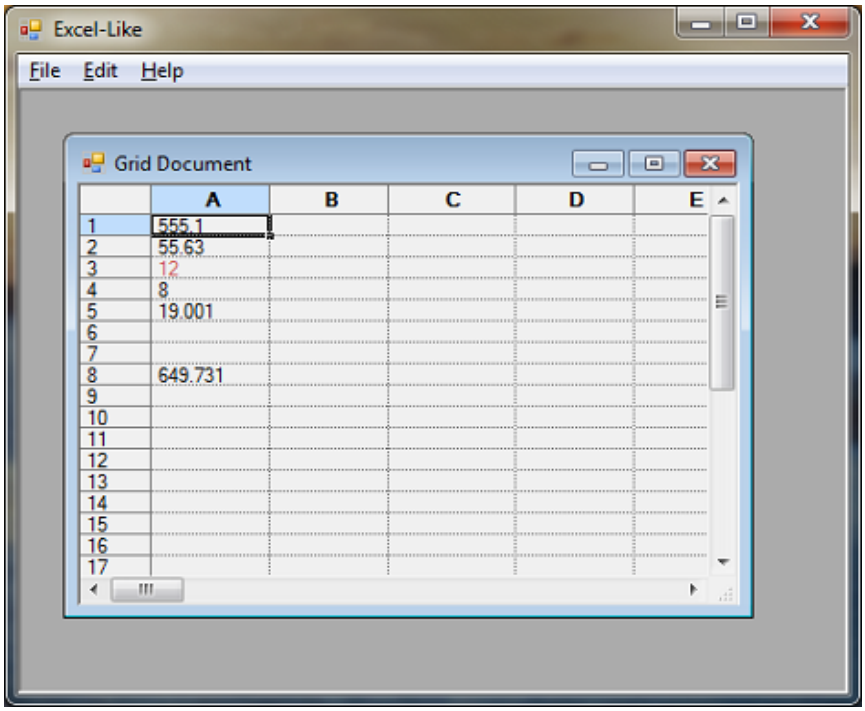
Find, Find and Replace

1. Select the first cell in column A1.
2. Select **Edit | Find**. Search for the number 7.

3. Select **Edit | Replace**. Search for the number 7 and replace every occurrence with the number 3. Before replacing, the grid looks like this:



After replacing all occurrences of 7 with 3, the grid looks like this:



The sum of 555.1, 55.63, 12, 8, and 19.001 is 649.731 not 649.331 as shown. The 7 in "649.771" was replaced with 3 after the evaluation of the formula.

NOTE >> If the value in a cell results from a formula and contains text that can be replaced, the formula is evaluated, the text is replaced, and then the resulting value replaces the formula.

Saving a Grid in HTML Format

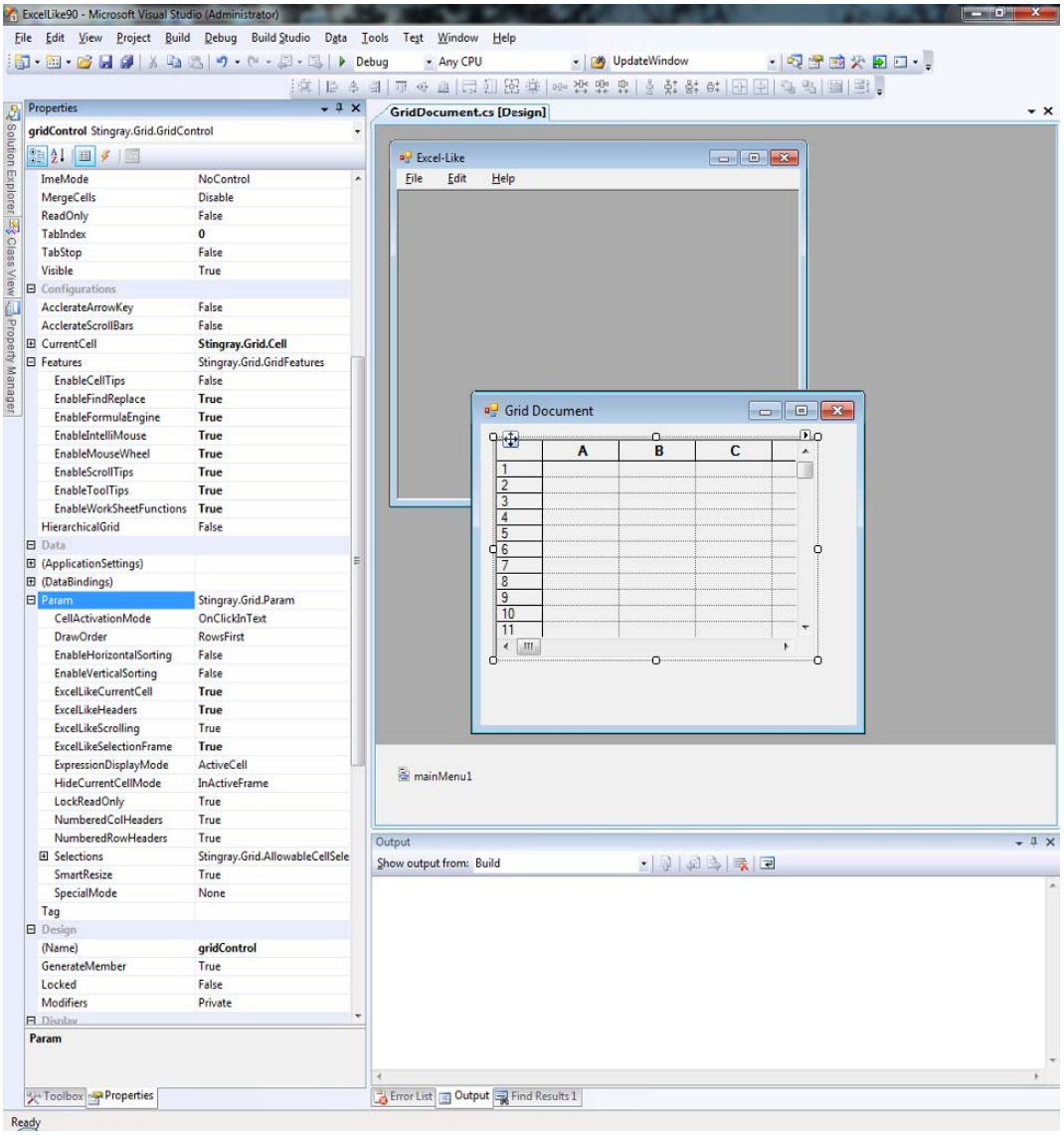
Saving grid data in HTML format is a means of presentation and does not preserve the data. You cannot restore the HTML format to another grid instance.

1. In cell A11, enter the formula `=sum(A1..A5)`.
2. Save the grid by selecting **File | Save As | HTML**. You are prompted for a filename.
3. View the file using a web browser. The formula (the one you stored in cell A11) is evaluated before the HTML view of the grid is constructed and stored. The formula remains unaltered in this sample.

For another example of saving in HTML format, see "RaceAttendance."

Design-time Features

Many Objective Grid for Microsoft .NET features can be set at design time using the property sheet designer:



You can hand-code these properties, but you may find it easier to let the designer help with the work. All available properties are enumerated, as well as their allowable values.

Setting Grid Properties

1. Select the GridDocument.cs designer window, and then select gridControl1 in the drop-down box at the top of the properties sheet.

2. Experiment with changing properties in these two property groups:
   - Configurations**->**Features—the formula engine related features
   - Data**->**Param—the Excel-like features
3. Experiment with changing other grid-wide properties, such as the number of row and columns in the grid, background color, default column width, layout and appearance, behavior, and accessibility.

Setting Cell Properties

1. Experiment with the Configurations**->**CurrentCell property group. This group allows you to set properties for the selected cell. The properties of a cell are determined by the style object contained by the cell.

Virtual Grid

Objective Grid for Microsoft .NET includes a sample virtual grid application. It is in the Samples subdirectory and is named Virtual Grid. The following steps explain how to create this project.

Create a solution for the virtual grid project

1. Start Visual Studio .NET.
2. From the **File** menu, select **File | New | Project**.
3. In the **Project Types** dialog, select **Visual C# Projects**.
   - For **Templates**, choose **Windows Application**.
   - For **Name**, enter **Virtual Grid**.
   - Select a **Location** on your hard drive.
4. Click **OK** to create the project. The Visual Studio .NET project wizard generates a project template according to your settings.

Place a GridControl on Form1.cs.

1. Select the **View | Solution Explorer** menu option to view the Solution Explorer.
2. In Solution Explorer, double-click `Form1.cs` to view the form in design mode.
3. Size the form as desired.
4. Select the **View | Toolbox** menu option to view the Windows Forms controls toolbox.
5. Drag a GridControl from the Toolbox onto the form.
6. Size the GridControl to fill the form.
7. With the GridControl selected in the form, set the following GridControl properties.
   - BorderStyle = `Fixed3D`
   - Anchor = `Top, Left, Bottom, Right`

Create a Data Structure for Storing Grid Cell Values

The class is implemented using a hash table, which hashes the cell values based on the row and column coordinates for the cell. The class implements ISerializable, so that it can be persisted. An indexer makes access to the grid data from our grid application more convenient.

1. Select the **View | Class View** menu option to view the class view for the project.
2. Right-click Virtual Grid project name, and then select **Add | Add Class** from the context menu.
3. In the C# Class Wizard:
   - For **Class Name**, enter DataArray.
   - Click **Finish**.
4. View the code for `DataArray.cs`. If it did not appear after you clicked **Finish**, double click `DataArray.cs` in the Solution Explorer, and then select the **View | Code** menu option.
5. Change the DataArray class definition so that it implements `ISerializable`:

```
public class DataArray : ISerializable
{
  public DataArray()
  {
    //
    // TODO: Add constructor logic here
    //
  }
}
```

6. At the top of the source file, add using declarations for `System.Collections` and `System.Runtime.Serialization`:

```
using System;
using System.Collections;
using System.Runtime.Serialization;
```

7. Add a private hash table data member to store the grid data:

```
public class DataArray : ISerializable
{
  private Hashtable data = new Hashtable();

  public DataArray()
  ...
```

8. Add code for an indexer to access the cell data by row and column indexes:

```
/// <summary>
/// This indexer allows a cell value to be hashed based on
/// the row and column indexes for the grid cell. A string
/// key is formed using the row and column indexes. The value
/// is then inserted into or obtained from the hash table
/// using this key.
/// The key is formed as, Row,Col. For example, cell 1,1
/// is hashed with the key, "1,1". Cell 10,5 is hashed with
```

```
/// the key, "10,5".
/// </summary>
public String this[int x, int y]
{
  get
  {
    // Form the hash key and an initially empty return
    // value string.
    String rc = "";
    String key = String.Format("{0},{1}", x, y);

    // If the data for the cell is present, then get it
    // from the table.
    if (data.Contains(key))
      rc = (String) data[key];

    return rc;
  }
  set
  {
    // Form the key
    String key = String.Format("{0},{1}", x, y);

    // If the table already contains the key...
    if (data.Contains(key))
    {
      // If the cell data is non-empty, store it
      if (value.Length > 0)
        data[key] = value;
      // Else if the new value is empty, remove
      // the key and value from the table
      else
        data.Remove(key);
    }
    // Else the table does not contain the key
    else
    {
      // Store the value only if it is non-empty
      if (value.Length > 0)
        data.Add(key, value);
    }
  }
}
```

The hash table key and value types are both .NET String instances. Only the cell values are hashed. The key is formatted using the row and column offsets. For example, the key for cell 1, 1 is `String(1,1)`.

9. The following two methods implement serialization for the grid cell data:

```
public virtual void GetObjectData(SerializationInfo s,
  StreamingContext c)
{
  s.AddValue("Data", data);
}

private DataArray(SerializationInfo s, StreamingContext c)
{
  data = (Hashtable) s.GetValue("Data", typeof(Hashtable));
}
```

10. The completed DataArray class should look something like the following:

```
using System;
using System.Collections;
using System.Runtime.Serialization;

namespace Virtual_Grid
{
  /// <summary>
  /// Summary description for DataArray.
  /// </summary>
  [ Serializable ]
  public class DataArray : ISerializable
  {
    private Hashtable data = new Hashtable();

    public DataArray()
    {
      //
      // TODO: Add constructor logic here
      //
    }

/// <summary>
/// This indexer allows a cell value to be hashed based on
/// the row and column indexes for the grid cell. A string
/// key is formed using the row and column indexes. The value
/// is then inserted into or obtained from the hash table
/// using this key.
/// The key is formed as, Row,Col. For example, cell 1,1
/// is hashed with the key, "1,1". Cell 10,5 is hashed with
/// the key, "10,5".
/// </summary>
    public String this[int x, int y]
    {
      get
      {
```

```
        // Form the hash key and an initially empty return
        // value string.
        String rc = "";
        String key = String.Format("{0},{1}", x, y);

        // If the data for the cell is present, then get it
        // from the table.
        if (data.Contains(key))
          rc = (String) data[key];

        return rc;
      }
      set
      {
        // Form the key
        String key = String.Format("{0},{1}", x, y);

        // If the table already contains the key...
        if (data.Contains(key))
        {
          // If the cell data is non-empty, store it
          if (value.Length > 0)
            data[key] = value;
          // Else if the new value is empty, remove
          // the key and value from the table
          else
            data.Remove(key);
        }
        // Else the table does not contain the key
        else
        {
          // Store the value only if it is non-empty
          if (value.Length > 0)
            data.Add(key, value);
        }
      }
    }

    public virtual void GetObjectData(SerializationInfo s,
      StreamingContext c)
    {
      s.AddValue("Data", data);
    }

    private DataArray(SerializationInfo s, StreamingContext c)
    {
      data = (Hashtable) s.GetValue("Data", typeof(Hashtable));
    }
  }
}
```

Add Virtual Grid Support to the Project

1. Add following using declarations:

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using Stingray.Grid;
```

2. Add a main menu for your application:

   — View `Form1.cs` in design view.

   — Drag a MainMenu from the toolbox onto the form.

   — Click on the menu text "Type Here". Add the following menu text: &File.

   — A new box appears below the new File menu. Add the following menu items to the File menu:

   &Save

   E&xit

   — Select the &Save menu item and change its properties as follows:

   Name = `FileSave`

   — Select the E&xit menu item and change its properties as follows:

   Name = `FileExit`

3. Add member variables for storing your data:

   — Select the **View | Code** menu option for Form1.

   — Create a new private DataArray attribute and a new Stream attribute.

```
private DataArray data = new DataArray();
private Stream s = null;
```

4. Add constructor code to load open, and load the DataArray containing the grid data:

```
//
// TODO: Add any constructor code after InitializeComponent call
//
s = File.Open("C:/temp/VirtualGrid.dat",
  System.IO.FileMode.OpenOrCreate,
  System.IO.FileAccess.ReadWrite);
BinaryFormatter bf = new BinaryFormatter();
if (s.Length > 0)
  data = (DataArray) bf.Deserialize(s);
```

Add Event Handlers for the Menu Commands and Virtual Grid Events

1. View `Form1.cs` in design view.
2. Select the GridControl in the designer by either selecting gridControl1 in the **Properties** list or by clicking any cell in the grid.
3. Click the lightening bolt in the grid properties window to view the Objective Grid for Microsoft .NET events.
4. Double-click the `GetStyleRowCol` event.
5. Add the following event handling code for the event:

```
private void gridControl1_GetStyleRowCol(object sender,
  Stingray.Grid.GetStyleRowColEventArgs e)
{
  if (e.Row > 0 && e.Col > 0)
  {
    e.Style.Value = data[e.Row, e.Col];
  }
}
```

6. View `Form1.cs` in design view.
7. Double-click the `StoreStyleRowCol` event.
8. Add the following event handling code for the event:

```
private void gridControl1_StoreStyleRowCol(object
  sender, Stingray.Grid.StoreStyleRowColEventArgs e)
{
  data[e.Row, e.Col] = e.Style.Value;
  e.Stored = true;
}
```

9. View `Form1.cs` in design view.
10. Click the **File** menu.
11. Double-click the **Save** menu option.
12. Add the following event handling code for this menu option:

```
private void FileSave_Click(object sender,
  System.EventArgs e)
{
  Cell cur = this.gridControl1.CurrentCell;
  data[cur.Row, cur.Col] = cur.ControlText;

  BinaryFormatter bf = new BinaryFormatter();
```

```
        s.SetLength(0);
        bf.Serialize(s, data);
      }
```

13. View `Form1.cs` in design view.
14. Click the **File** menu.
15. Double-click the **Exit** menu option.
16. Add the following event handling code for this menu option:

```
      private void FileExit_Click(object sender,
        System.EventArgs e)
      {
        this.Close();
      }
```

17. Compile and run your virtual grid application.

RaceAttendance

This sample illustrates properties of the Stingray.Grid.Style class by formatting and analyzing data from a series of races. It also demonstrates:

- Printing
- Print preview
- Saving grid data to an HTML file
- Event handling
- Embedding a properties sheet, so that the grid and cell properties can be manipulated at runtime

Some grid-wide properties can be set only at design time. If you attempt to set them at runtime, an error message is displayed.

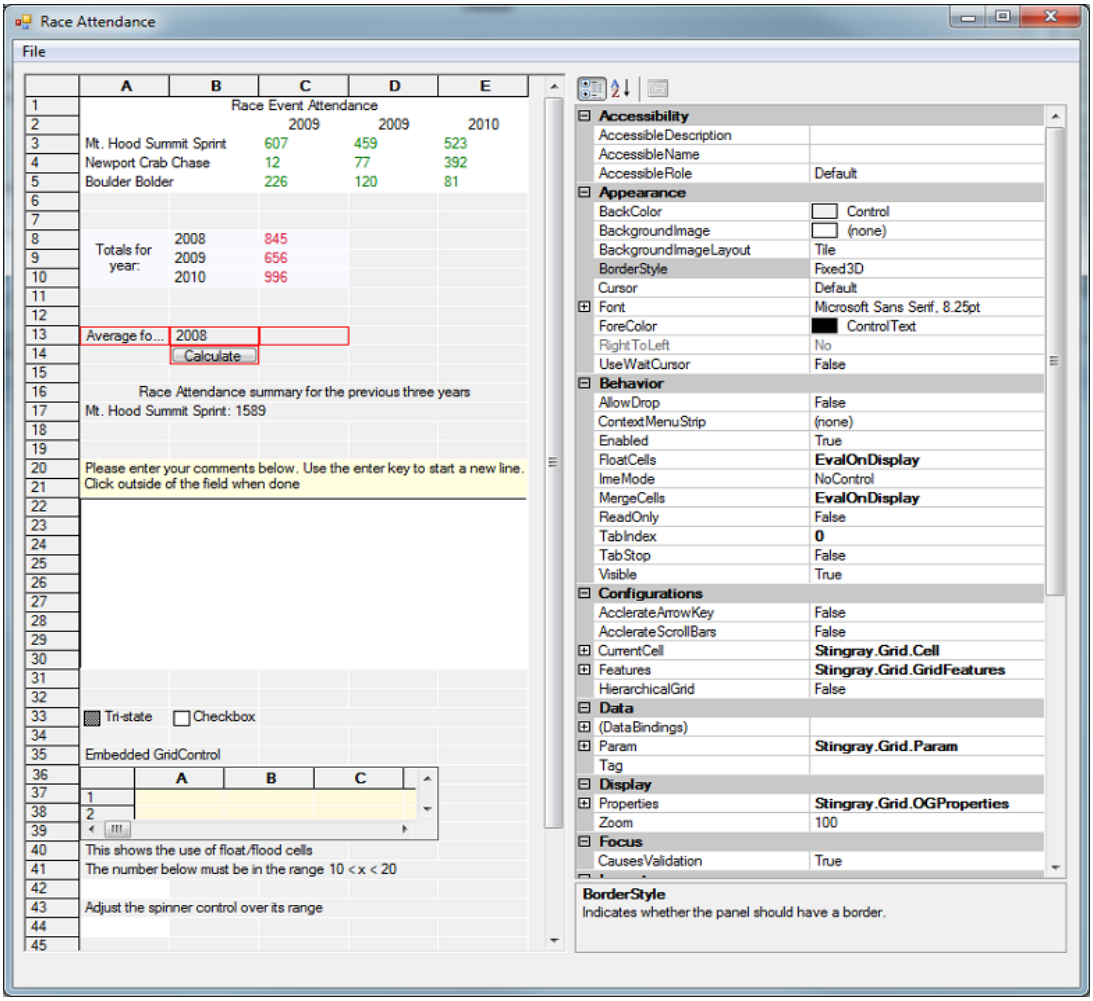
At runtime, as you select each cell, the Configurations**->**CurrentCell group in the properties sheet changes to reflect the properties of the selected cell. You can see and experiment with the properties.

Nearly the entire grid is constructed by code contained within the `Form1.FormLoad()` method. Look at the source code as you read through this sample. Each major section of the code is identified by `// Section #: ...`, which matches the following sections.

The application has four major components:

- Stingray.Grid.GridControl (called “grid”)
- System.Windows.Forms.PropertyGrid (called “property grid”)
- A menu
- A System.Windows.Forms.Form (called “form”), on which the first three components are embedded

Figure 42 – RaceAttendance Form Design



Connecting the Property Grid to the Grid

You can easily embed a grid and property grid in the same form by dragging and dropping both components onto a form, but you need to tie them together so that changes in one are reflected in the other. This is accomplished with a single line:

```
propertyGrid1.SelectedObject = gridControl1;
```

Adding Data to the Grid

The second section of code formats part of the grid to present the raw data used by the rest of the sample program. It used properties and attributes such as background color, merged cells, and alignment.

1. A Stingray.Grid.Range object is created and refers to a rectangular region of cells from A1 to E5. Because the range object uses integer indices, A1 translates to the coordinate (1,1) and E5 translates to (5,5). The cells in this range are painted with white as a background color and serve as a table of raw values to be used elsewhere.

2. The merged cells technique is used to create a center-aligned title for the table. To enable the `MergedCells` property of the grid:

   ```
   gridControl1.MergeCells = MergeCellsMode.EvalOnDisplay;
   ```

   The title spans the width of the grid and is positioned in the center by filling all five cells with the same value (`Race Event Attendance`), and by setting the `MergeCell` and `HorzAlign` properties of each cell.

3. To make the process more efficient, a Style object is created, the appropriate properties set, and then applied to the range A1 through A5 (grid coordinates (1,1) through (1,5)) with the call:

   ```
   gridControl1.SetStyleRange(headerRange, headerStyle);
   ```

   Each cell in the range A1..A5 has exactly the same value, but only one copy of the title is displayed because the cells are merged.

4. To keep the title from being inadvertently changed:

   — The `ReadOnly` property, which prevents the Style properties from being changed, is enabled

   — The `Enabled` property is disabled, which prevents the cells from being selected

5. Column titles (such as 2001) and rows names (such as Mt. Hood Summit Sprint) are added. The Style applied to each row title has the `EllipseType` set to `EllipseType.DotEllipse`. When you run the application for the first time, you may not see any evidence of "..." in the row titles. You can force the ellipses to appear if you make the 'A' or 'B' columns narrower.

6. Additional Style attributes applied to the row titles are `Enabled` (set to `false`) and `ToolTip` (set to the same value as the text). Setting `Enabled` to `false` stops you from selecting the cell(s). Setting `ToolTip` to the same value as the text causes a pop up tag to display the full cell text, which is useful if some of the cell is obscured.

7. The table of raw data is filled in. The Style applied to each of these cells includes green text and allows only numeric values.

Formulas

The third section shows how to store formulas in a cell.

If you want to store a simple value such as a string or a numeric, do something like this:

```
gridControl1[3,7].Style.Value = "This is a test";
```

or this:

```
Style s = new Style();
s.Value = "12.34";
s.TypeOf = StyleValueType.typeNumeric;
gridControl1[4,10].Style = s;
```

Storing a formula or an expression is a bit different. You *cannot* do this:

```
gridControl1[3,7].Style.Value = "=A1+A2+A3";
gridControl1[3,7].Style.TypeOf = StyleValueType.typeExpression;
```

or this:

```
Style s = new Style();
s.Value = "=A1+A2+A3";
s.TypeOf = StyleValueType.typeExpression;
gridControl1[4,10].Style = s;
```

NOTE >> This behavior may change in future versions of Objective Grid for Microsoft .NET.

To store formulas, use the `SetExpressionRowCol` method of the GridControl class:

```
gridControl1.SetExpressionRowCol( 8,3,"=C3+C4+C5" );
```

The expressions stored are of the form `=X1+X2+X3`. You could just as easily use the form `=X1..X3`.

Controls and Event Handling

The fourth section calculates the average for a selected year.

1. Select a year from the drop-down box in cell B13 (which may not be visible unless the cell is active). The average for the selected year is placed in the adjacent cell.

   The code uses control embedding and event handling —responding to events from a control embedded in a cell. In this case, a push button triggers the calculation based upon a value selected in another control.

   The push button control in cell B14 is tied to the private method `DispatchLClick()`, which is called in response to the event `Stingray.Grid.LButtonClickedRowCol`. The association of the `LButtonClickedRowCol()` and the method `DispatchLClick()` is made by selecting the events list for the GridControl in the property sheet.

2. Locate the `LButtonClickedRowCol` event and enter the method name to associate with the event. Once that is done, you can write in the code to test which cell (via row and column) originated the event and respond accordingly.

Miscellaneous

The last section demonstrates features that are not immediately obvious.

Creating a Title

Cell A16 illustrates a better way of creating a title that spans several cells. The first technique ("Adding Data to the Grid") put the same string in several cells, and set the `MergeCell` property in each of those cells. The following technique does not require storing multiple instances of a string.

- Identify which cells to cover with the title, and then call the `SetCoveredCellsRowCol()` method of the GridControl class. This allows a value in one cell to flow over following cells (for example, the contents of cell A16 can cover cells A17 and A18). The dividing line between the cells is hidden, giving the appearance that cells are one big cell.

  Calling `SetCoveredCellsRowCol()` is not always sufficient. If the title text is too short, only some of the rows are covered. To complete the effect, set the `Style.HorzAlign` property to `Center`.

  NOTE >> Cell A17 is a spin control over three lines of data. You may have to select the spin control text to reveal the spin control buttons. This issue will be addressed in a subsequent release.

- Rows 20 and 21 show text in a cell overflowing cells in multiple rows.
- Rows 22 through 30 are set up as a multi-line edit control that accepts user input, including newlines.
- Row 33 shows normal and tristate check boxes.
- Row 34 is an illustration the flood and float cell style properties.
- Cell A36 is an entry field that accepts only numeric values in the range 10<x<20.
- Cell A38 is another spin control with a fixed range of 1 through 10.
- Cell A42 is another entry field that accepts up to a fixed number of characters.

Chapter 10

Working with the Office Open XML 2.5 SDK

Office Open XML SDK for Excel

Office Open XML is an open and standardized XML format for Office 2007 or higher file, based on the following standards:

- ECMA-376
- ISO/IEC 29500-1:2008

These standards define the structure and the elements for Office Open XML files. The Office Open XML files themselves, i.e. `*.xlsx` for Excel, are zipped archives that contain a specific directory and file structure. The files that hold the content of a spreadsheet are XML files like any other XML files.

The basic Office Open XML file components are:

- **Package:** The zipped archive itself, e.g., `filename.xlsx`
- **Parts:** The files inside the package holding the content. Most of these are XML files, but other files types, such as binary, are possible.
- **Content types:** Each part has an associated content type that is enforced on open.
- **Relationships:** Associations between the package and its top-level parts, and between parent parts and child parts.

This table provides an example of the XML structure and files in an Excel `.xlsx` file.

Table 38 – Example Excel .xslx file structure

`_rels`	Relationships among XML components
`xl/_rels`	Relationships among XML components
`xl/drawings/drawing<1...n>.xml`	Drawing components used in the worksheets
`xl/worksheets/sheet<1...n>.xml`	The worksheet data files
`xl/sharedStrings.xml`	Strings shared among worksheet cells
`xl/styles.xml`	Styles used in the worksheets
`xl/workbook.xml`	Sheets, controls, and other components that are used in the workbook
`[Content_Types].xml`	Defines parts and extensions for the spreadsheet

The Office Open XML 2.5 SDK provides an API to help in the direct interaction with XML elements between an application and `xlsx` files, without Excel or any other Office application having to be installed on the development or deployment machine. The following sections describe the new reference assembly for Objective Grid for .NET, and a sample that illustrates the use of the API.

OOXML Excel Library Addition

A new assembly, GridExcelHelper<*compiler_version*>, has been added to the Objective Grid for .NET set of assemblies to assist in the reading and writing of Excel 2007 or higher `xlsx` files using the Open Office XML SDK. The GridExcelHelper library references the following assemblies in addition to the common System assemblies:

- DocumentFormat.OpenXml — Open Office XML
- Stingray.GridControl<*compiler_version*> — Objective Grid for .NET

The GridExcelHelper project is broken down into the following C# files:

- GridExcel.cs
- ExcelFiles.cs

The common namespace used throughout the GridExcelHelper assembly is `Stingray.Excel`.

GridExcelHelper is the base class for the assembly. It is defined in `GridExcel.cs`. This file contains the public API for all functions that deal with the contents of an XLSX file and the grid control. This class contains the following public API, presented in category order:

Public data members

```
enum WarningReason{ OpenFailed, SaveFailed, OpenRenamed, SaveRenamed }
enum WarningReason{ Open, Save }

struct CellIndex containing
    int nRow
    int nCol
```

Public API (non-virtual)

```
void AddEditedCell(int nRow, int nCol)
void CopyToGrid(ExcelFile excelFile, out int rowMax, out int colMax)
string GetFileName(FileDialogMode mode)
void Init()
string LoadGridFromExcelFile(string newFileName, string sheetName,
  out int rowMax, out int colMax)
string SaveGridToExcelFile(string newFileName, string sheetName)
```

The public API functions do the following:

- **AddEditedCell** keeps a list of all modified cells.
- **CopyToGrid** copies the data from each cell reference to the grid control and then closes the current spreadsheet.
- **GetFileName** launches either the OpenFileDialog or the SaveFileDialog to obtain a filename based on the mode passed to it.
- **LoadGridFromExcelFile** loads the given `xlsx` file into the grid control.
- **SaveGridToExcelFile** saves all the grid control cell modifications to the given Excel `xlsx` file and sheet.

Public API (virtual with a default implementation)

```
virtual void UpdateExcelFromGrid(Sheet sheet, ExcelFile excelFile)
```

Default implementation details may be viewed in the source file `<StingrayInstalldir>\Objective Grid for Microsoft .NET\Solutions\GridExcelHelper\GridExcel.cs`.

The functions `ClearGrid`, `SetGridValue`, `GetGridCellValue` and `Warning` contain a `Debug.Assert` statement as the default implementation. These functions are left to the user to define.

Implementation definition examples can be seen in the tutorial sample `<StingrayInstalldir>\Objective Grid for Microsoft .NET\Tutorials\ExcelReadWrite_CS`.

ExcelFile is a base class for the assembly. It is defined in `ExcelFile.cs`. This file contains the public API for all functions that deal with the `xlsx` file. This class contains the following public API, presented in category order:

Properties

```
string failedFileName { get; set; }
Stream currentFileStream { get; set; }
string currentSheetName { get; set; }
SpreadsheetDocument currentSpreadSheet { get; set; }
WorkbookPart currentWorkbookPart { get; set; }
Worksheet currentWorkSheet { get; set; }
```

Public API

```
ExcelFile()
static Boolean IsFileLocked(FileInfo file)
string ColName(string CellReference)
static string ColumnNameFromIndex(uint columnIndex)
int ExcelColumnNameToNumber(string columnName)
int RowNumber(string CellReference)
void UpdateCell(Sheet sheet, string value, int nRow, int nCol)
bool UpdateValue(Sheet sheet, string cellRef, string value, bool isString)
string ExcelGetCellValue(Sheet theSheet, string cellRef)
```

The public API functions do the following:

- `ExcelFile` is the constructor for this class.
- `IsFileLocked` uses a flag to determine if the specified file is being used or not. If the specified file is being processed or is not found, `IsFileLocked` returns `TRUE`.
- `ColName` extracts the column's cell reference and returns the column name as a string.
- `ColumnNameFromIndex` converts a column number to a column name, i.e. A, B, C, ..., AA, AB, ...
- `ExcelColumnNameToNumber` converts the column's string name to a 32-bit integer value.
- `RowNumber` converts the row's string cell reference to a 32-bit integer and returns the numeric row number.
- `UpdateCell` calls `UpdateValue`.
- `UpdateValue` either retrieves the index of an existing string or inserts the string into the shared string table. Then `UpdateValue` gets the index of the new item and saves the worksheet.
- `ExcelGetCellValue` retrieves a reference to the worksheet part, and then use its worksheet property to get a reference to the cell whose address matches the address supplied. If the cell exists, the cell's value is returned. Otherwise, if the cell does not exist, an empty string is returned.

Public API (virtual with default implementation)

```
virtual Sheet CopyFileAndOpen(string fileName, string sheetName)
virtual Sheet OpenFile(string fileName, string sheetName)
virtual Sheet Save(string fileName, string sheetName)
virtual Sheet SaveAs(string fileName, string newFileName, string sheetName)
```

Here is what the supplied default implementations do. You can override them, of course.

- `CopyFileAndOpen` copies the Excel `xlsx` file to a new `xlsx` file and then opens the copy.
- `OpenFile` opens the current Excel `xlsx` file if it is not locked. Otherwise, `CopyFileAndOpen` is called.
- `Save` opens the original Excel `xlsx` file, updates all cell references, and then saves the file.
- `SaveAs` opens the original Excel `xlsx` file, updates all cell references, then saves the file under a new filename as defined by the user.

Default implementation details may be viewed in the source file `<StingrayInstalldir>\Objective Grid for Microsoft .NET\Solutions\GridExcelHelper\ExcelFile.cs`.

OOXML Excel Library Example

To illustrate the use of this new assembly, we've created a new tutorial sample called, `ExelReadWrite_Step1_<compiler_version>`. This project is located in `<stingray-installdir>\Objective Grid for Microsoft .NET\Tutorials\ExcelReadWrite_CS`.

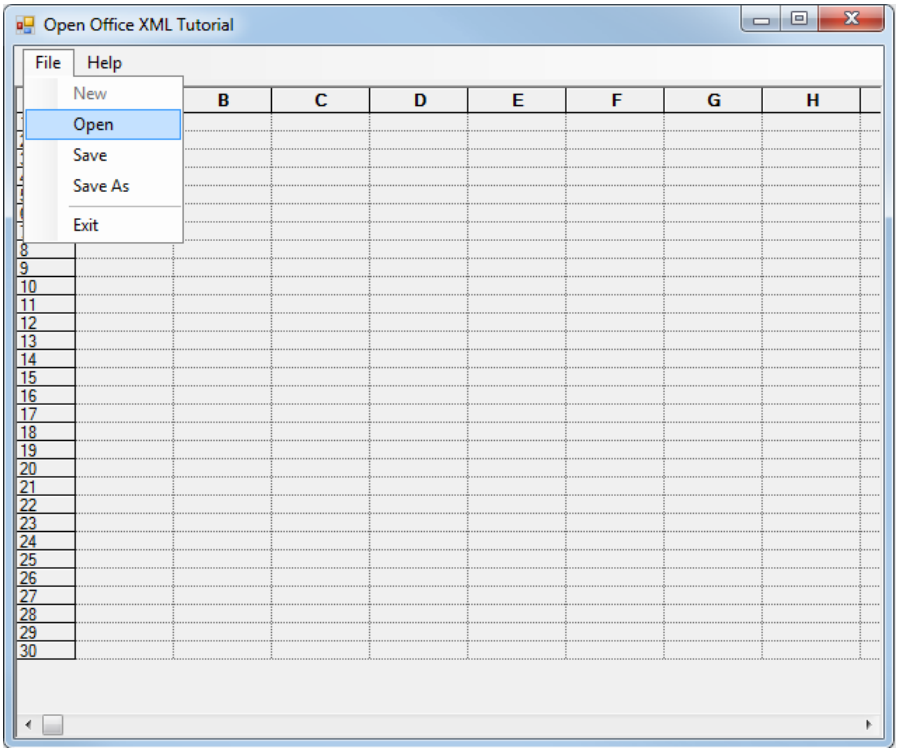
This sample requires the following assembly references in addition to the common System assemblies:

- DocumentFormat.OpenXML
- Stingray.GridControl<*compiler_version*>
- Stingray.GridUtils<*compiler_version*>
- Stingray.GridExcelHelper<*compiler_version*>

The sample illustrates the following Open Office XML operations for an Excel 2007 or higher `xlsx` file:

- Open or read an existing `xlsx` file into a GridControl on a WinForm
- Save changes to an existing and open `xlsx` file
- Save as a new `xlsx` file

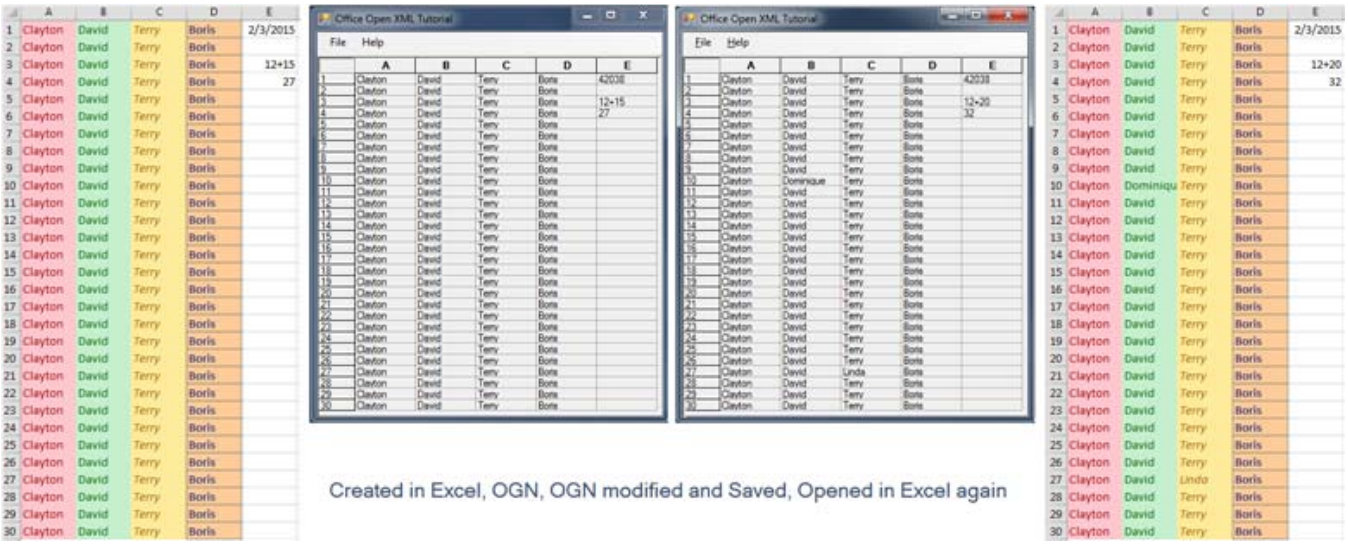
When built successfully, the running sample looks like this:



The `xlsx` files should be able to be read from Excel 2007 or higher into an Objective Grid for .NET control on a WinForm. This same file, whether from a modified Excel 2007 or higher `xlsx` file or saved to a new Excel 2007 or higher `xlsx` file, should also be capable of being read from Excel 2007 or higher.

Data modification from and to an `xlsx` file is available. Data types include numbers as integers, strings, and dates. Formatting and formulae are not interchangeable between the GridExcelHelper assembly and Excel 2007 or higher. For example, if an Excel spreadsheet is created with colored cells, the colors will not appear in the grid control. Also, if an Excel spreadsheet contains a formula, the result (not the formula) will appear in the grid control.

However, every effort has been made to preserve existing Excel formatting and formulae in the Save or Save As implementation. The `xlsx` XML contains all the original information while only the data values modified within the grid control are saved out to the `xlsx` and XML file structures. This allows the Excel-generated spreadsheet to maintain it's appearance and values as illustrated below:



Index

Symbols

.NET cell controls 99
- conserving memory 101
- limitations 103
- monitoring behavior 101
- monitoring cell editing 102
- style manipulation code 101
- using 104
- using as cell editors 101

.NET functionality, extending 76

Numerics

1stGrid tutorial 120

B

background color, matching in a grid 89

base styles
- definition 4

C

CanceledEditing event 105

CancelEditing event 105

CanStore property 98

CellControl class 99

cells
- changing attributes 61
- computing coordinates 71
- covering 63
- definition 4
- definition of covered 4
- definition of current 4
- disabling 62
- merging 63
- modifying attributes 61
- reading data from 55
- readonly 63
- writing data to 55

changing
- attribute of a cell 61
- the number of rows and columns 55

Col method 119

Col2 method 119

Col3 method 119

color printing, enabling 91

columns
- changing the number of 55
- deleting 67
- freezing 69
- hiding 67
- inserting 66

common terms 4–5

control
- definition 4

controlling
- size of virtual grids 98

covered cells
- definition 4

covering
- cells 63

CreateChildAt method 115

current cell
- definition 4

D

data source
- definition 4

DeleteChildAt method 114

DeleteChildren method 114

deleting
- rows and columns 67

disabling cells 62

documentation
- formats 5

DrawCellControl event 105

DrawInactiveCell property 104

E

Enable method 118

EnableHierarchicalGrid
- method 114

EndEditing event 105

Excel-like workbook interface 108

ExpandRow method 114

F

freezing
- rows and columns 69

G

GetChildAt method 114

GetParent method 114

glossary 4

Grid property 104

grid type, how to choose 52–53

GridControl class 112
- about 112

GridControl.RegisterControl
- method 99

grids
- hierarchical 112
- line display and printing 91
- matching background colors 89
- reading from 55
- virtual 95
- writing to 55

GridTabControl class
- about 107
- using 111

GridTabControlDesigner class 110

H

HgMap property 114, 118, 119

HideCurrentCell event 105

HideExpanderColumn method 114

hiding
- rows and columns 67

hierarchical grids
- about 112
- features 113
- logical cell coordinates 118
- methods 115
- physical cell coordinates 118
- properties 114

using 116
HierarchicalGrid property 114
HierGridMap class 118

I
InitCurrentCell event 105
InitializeChildAt method 115
inserting
rows and columns 66
InstantiateChildAt method 115
IsChildShown method 114
IsRowExpandable method 115

L
LeftCell event 105
LtoP.Col method 118
LtoP.Range method 118
LtoP.Row method 118

M
merging cells 63
MFC libraries 76
ModifyCell event 105
modifying
attribute of a cell 61

P
print margins, setting 83, 90
printing grid lines 91
printing, enabling color 91
ProcessHorizontalArrows
property 104
ProcessVerticalArrows
property 104
properties
CanStore 98
definition 5
DrawInactiveCell 104
Grid 104
HgMap 114, 118, 119
HierarchicalGrid 114
ProcessHorizontalArrows 104
ProcessVerticalArrows 104
SelectedIndex 109
SelectedSheet 109
Stored 98
Style.CustomControl 99
TabIdx 109
TabLabel 109
UseCellStyle 104
PtoL.Col method 118
PtoL.Range method 118
PtoL.Row method 118

R
range
definition 5
Range method 119
reading
from a grid 55
readonly cells 63
RefreshCurrentCell event 105
removing
rows and columns 67
ResetCurrentCell event 105
Resume method 118
Row method 119
Row2 method 119
Row3 method 119
rows
changing the number of 55
deleting 67
freezing 69
hiding 67
inserting 66

S
samples 144
ExcelLike 147
RaceAttendance 164
SimpleSpreadSheet 146
Virtual Grid 156
SelectedIndex property 109
SelectedSheet property 109
StartEditing event 105
StoreCurrentCell event 106
Stored property 98
storing
data in the grid 55
style
definition 5
Style.CustomControl property 99
Suspend method 118

T
TabIdx property 109
TabLabel property 109
terms,common 4
tutorials
1stGrid 120

U
UseCellStyle property 104
using
.NET cell controls as cell
editors 101
virtual grids 97

V
ValidateCell event 106
virtual grids
about 95
advanced topics 98
controlling the size 98
customizing behavior 98
reasons for using 96
using 97
Visual Studio .NET
version compatibility 1

W
workbook
definition 5
worksheet
definition 5
writing
to a grid 55