ComponentOne

FlexSheet for WPF

GrapeCity US

GrapeCity
201 South Highland Avenue, Suite 301
Pittsburgh, PA 15206
Tel: 1.800.858.2739 | 412.681.4343
Fax: 412.681.4384
Website: https://www.grapecity.com/en/
E-mail: us.sales@grapecity.com

Table of Contents

FlexSheet for WPF Overview

FlexSheet for WPF is a fast, lightweight, and powerful control that extends the functionality of **C1FlexGrid** to provide many Excel-like features. It provides built-in features such as worksheet management, a powerful formula engine, support style format and many more. You can easily import and export spreadsheets, apply formulas, manipulate rows and columns, insert charts and images, perform undo and redo operations, filter columns, format cells and perform cell styling.

C1FlexSheet is a new spin to an old classic, **C1FlexGrid**. What makes it different from **C1FlexGrid** is the function support which makes it behave like a spreadsheet.

Help with WPF and Silverlight Edition

For information on installing **ComponentOne Studio WPF and Silverlight Edition**, licensing, technical support, namespaces, and creating a project with the controls, please visit Getting Started with WPF Edition.

FlexSheet Key Features

In addition to providing full functionality of the **FlexGrid** control, **C1FlexSheet** provides many more features. Make the most of **FlexSheet for WPF** by taking advantage of the following key features:

View and Edit Excel Files

FlexSheet for WPF allows you to import, edit, and export the Microsoft Excel files (.xls, .xlsx). FlexSheet manages multiple worksheets with an Excel-like tabbed interface. You can switch between sheets, edit the content directly on a sheet, and update the content via code.

Manage Multiple Sheets Tabbed UI

FlexSheet for WPF provides the ability to manage multiple sheets in a single tabbed User Interface (UI). Users can select, add, remove, move, and rename the sheets, and also perform the same actions from code.

Insert Charts, Images, Sparklines and Comments

FlexSheet for WPF offers the ability to add comments and insert floating objects such as charts, sparklines and images to your worksheets.

Formula Support

FlexSheet for WPF is equipped with a powerful formula engine to analyze and evaluate Excel-style formulas. Several built-in functions are supported such as Aggregate, Statistics, and Trigonometric functions, and references to other sheet/cell values can also be included. When the entered text starts with "=", it will be recognized as a formula. The formula is analyzed by FlexSheet, and the result of the formula will be displayed in the cell.

Styles and Formatting

FlexSheet for WPF provides the ability to customize cell styles including font, color, size, and borders. You can specify the display format, such as numbers, currency, percentage and date. Easily apply styles and formatting to the whole sheet including rows, columns, cell and cell range, to customize your app's appearance.

Manage Worksheets

Easily manipulate sheets contained in a workbook - create new sheets, delete sheets, show/hide sheets, and change the tab order. Also, you can add, delete, and fix the rows/columns in a sheet, and show/hide header and grid lines.

Quick Start: FlexSheet for WPF

The quick start guide familiarizes you with some of the features of **FlexSheet for WPF**. In this section, you learn to create a new WPF project in Visual Studio, add C1FlexSheet control to the application, and populate data to understand the appearance and working of the control.

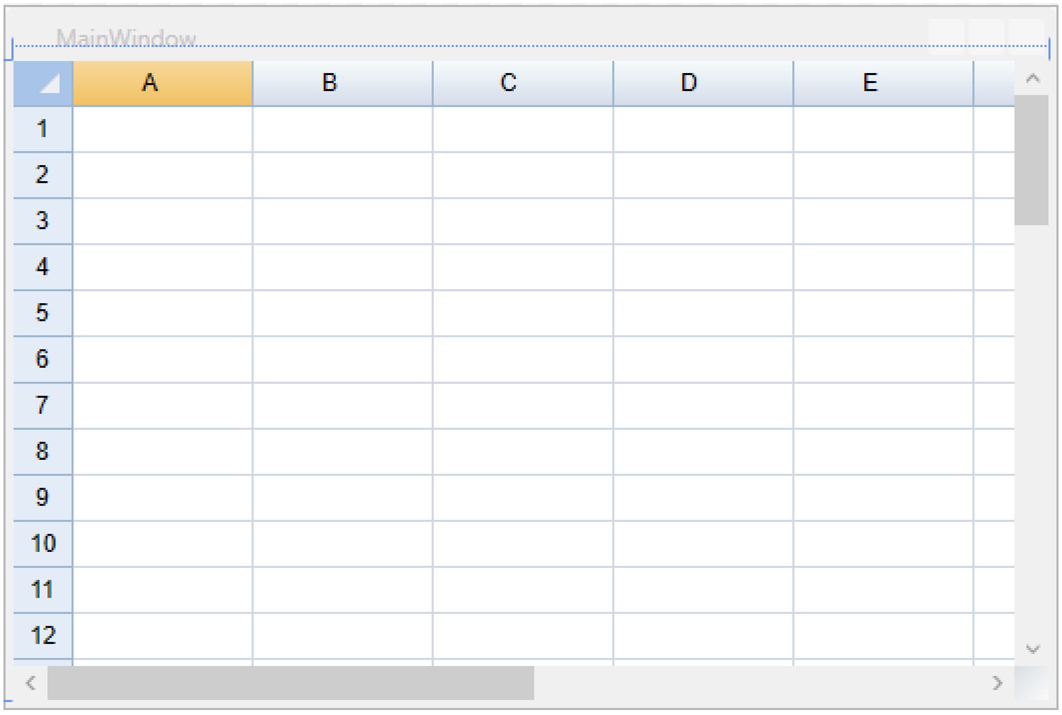
Step 1 of 3: Setting up the Application

In this step, you begin with creating a WPF application in Visual Studio and then adding a FlexSheet control to your application.

In Design View

To add a FlexSheet to your WPF application in Design view, perform the following steps:

1. Create a new WPF project in Visual Studio.
2. Navigate to the Toolbox and locate the C1FlexSheet control icon.
3. Double-click the C1FlexSheet icon to add the control to the MainWindow. The control looks like the following:



In Code

To add a FlexSheet to your WPF application in Code view, perform the following steps:

1. Set the Name property of the Grid in XAML so that the control has a unique identifier to call in code. In our case, Name property of the Grid control is set to Parent, as shown in the following code:

XAML copyCode
```
<Grid Name="Parent">
</Grid>
```

2. Add the following namespaces in Code view:

- **Visual Basic**

```
Imports C1.WPF
Imports C1.WPF.FlexGrid
```

- **C#**

```
using C1.WPF;
using C1.WPF.FlexGrid;
```

3. Add the following lines of code beneath the **InitializeComponent()** method to add the FlexSheet control:

- **Visual Basic**

```
Dim flex = New C1FlexSheet()
Parent.Children.Add(flex)
```

- **C#**

```
var flex = new C1FlexSheet();
Parent.Children.Add(flex);
```

Step 2 of 3: Adding a Sheet and Populating it with Data

In the last step, you created a WPF application and added the C1FlexSheet control to it. In this step, you add a sheet to the control and populate it with ordered data to see how the control works.

1. Add a new sheet to **FlexSheet** control by adding the following code beneath **InitializeComponent()** method in the interaction logic for XAML:

- **Visual Basic**

```
flex.AddSheet("Sheet1", 50, 10)
```

- **C#**

```
flex.AddSheet("Sheet1", 50, 10);
```

2. Populate the sheet with data by adding the following code:

- **Visual Basic**

```
' populate the grid with some formulas (multiplication table)
For r As Integer = 0 To flex.Rows.Count - 3
    Dim datas As New List(Of Double)()
    For c As Integer = 0 To flex.Columns.Count - 1
        flex(r, c) = String.Format("={0}*{1}", r + 1, c + 1)
        Dim value As Double = CDbl(flex(r, c))
        datas.Add(value)
    Next
```

- **C#**

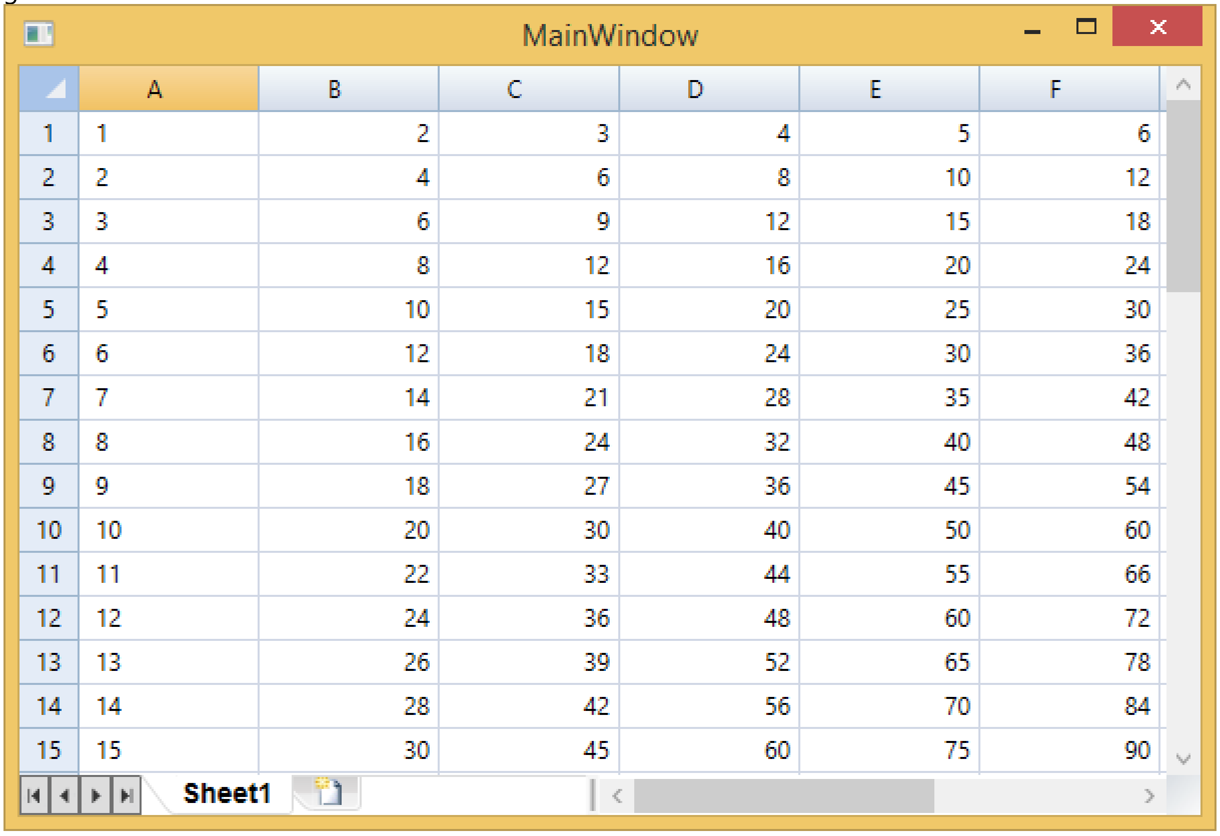
```
// populate the grid with some formulas (multiplication table)
for (int r = 0; r < flex.Rows.Count - 2; r++)
{
    List<double> datas = new List<double>();
    for (int c = 0; c < flex.Columns.Count; c++)
    {
        flex[r, c] = string.Format("={0}*{1}", r + 1, c + 1);
        double value = (double)flex[r, c];
        datas.Add(value);
    }
}
```

With this, you have successfully added a sheet to your **FlexSheet** control and populated it with data.

Step 3 of 3: Running the Application

In previous step, you added a sheet to the C1FlexSheet control and populated it with data. In this step, you will run the application and view populated data in FlexSheet.

Press **F5** to run the application and view the data in FlexSheet control. FlexSheet control will look similar to the image given below:



	A	B	C	D	E	F
1	1	2	3	4	5	6
2	2	4	6	8	10	12
3	3	6	9	12	15	18
4	4	8	12	16	20	24
5	5	10	15	20	25	30
6	6	12	18	24	30	36
7	7	14	21	28	35	42
8	8	16	24	32	40	48
9	9	18	27	36	45	54
10	10	20	30	40	50	60
11	11	22	33	44	55	66
12	12	24	36	48	60	72
13	13	26	39	52	65	78
14	14	28	42	56	70	84
15	15	30	45	60	75	90

Customizing Appearance

C1FlexSheet is designed to make customizations easy for you. You can change the appearance of tab strips, format cells, and include charts, images and sparklines in the control to change its look and feel. The topics listed below provide information on the customizable elements within C1FlexSheet.

Using Floating Objects

FlexSheet for WPF can be customized by adding floating objects such as charts, sparklines and images to your C1Flexsheet control. You can insert charts and sparklines in C1FlexSheet to compare and visualize data and also insert images to enhance the presented data. For more information, refer Inserting Charts, Inserting Sparkline, and Inserting Images.

Inserting Charts

The data in C1FlexSheet can be visualized using Charts. To visualize the data in charts, you need to add data to a sheet in C1FlexSheet.
Follow the given steps in XAML to add data to a sheet and visualize it in chart:

1. Add the following namespace declaration in the **Window** tag:

XAML
```
xmlns:c1="http://schemas.componentone.com/winfx/2006/xaml"
```

2. Copy the following code inside **Grid** tag to add C1FlexSheet control and a button control to insert a chart:

XAML copyCode
```
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition/>
</Grid.RowDefinitions>
    <c1:C1FlexSheet Name="flex" Grid.Row="1" HorizontalAlignment="Left"
VerticalAlignment="Top"/>
    <Button Content="Insert Chart" Name="btnInsertchart" Click="btnInsertchart_Click"
HorizontalAlignment="Left" Width="75"/>
```

3. Right-click Design view and select **View Code** from the context menu.
4. Add the following namespaces at the top:
   - **Visual Basic**
```
Imports C1.WPF
Imports C1.WPF.FlexGrid
Imports C1.WPF.C1Chart
```
   - **C#**
```
using C1.WPF;
using C1.WPF.FlexGrid;
using C1.WPF.C1Chart;
```
5. Insert the following code directly below the **InitializeComponent()** method to add a sheet and data in it:
   - **Visual Basic**
```
'adding Sheet
flex.AddSheet("Sheet1", 50, 10)

' populate the grid with some formulas (multiplication table)
For r As Integer = 0 To flex.Rows.Count - 3
    Dim datas As New List(Of Double)()
    For c As Integer = 0 To flex.Columns.Count - 1
        flex(r, c) = String.Format("={0}*{1}", r + 1, c + 1)
        Dim value As Double = CDbl(flex(r, c))
        datas.Add(value)
    Next
```

- **C#**

```
//adding Sheet
flex.AddSheet("Sheet1", 50, 10);

// populate the grid with some formulas (multiplication table)
for (int r = 0; r < flex.Rows.Count - 2; r++)
{
    List<double> datas = new List<double>();
    for (int c = 0; c < flex.Columns.Count; c++)
    {
        flex[r, c] = string.Format("={0}*{1}", r + 1, c + 1);
        double value = (double)flex[r, c];
        datas.Add(value);
    }
}
```

6. Go back to the Design view and select **Event handlers** of **btnInsertchart** from the **Properties** windows.
7. Double-click the **btnInsertchart_Click** event handler.
   The Code view will open again.
8. Add the following code to the **btnInsertchart_Click** event handler to insert a chart on button click:
   - **Visual Basic**

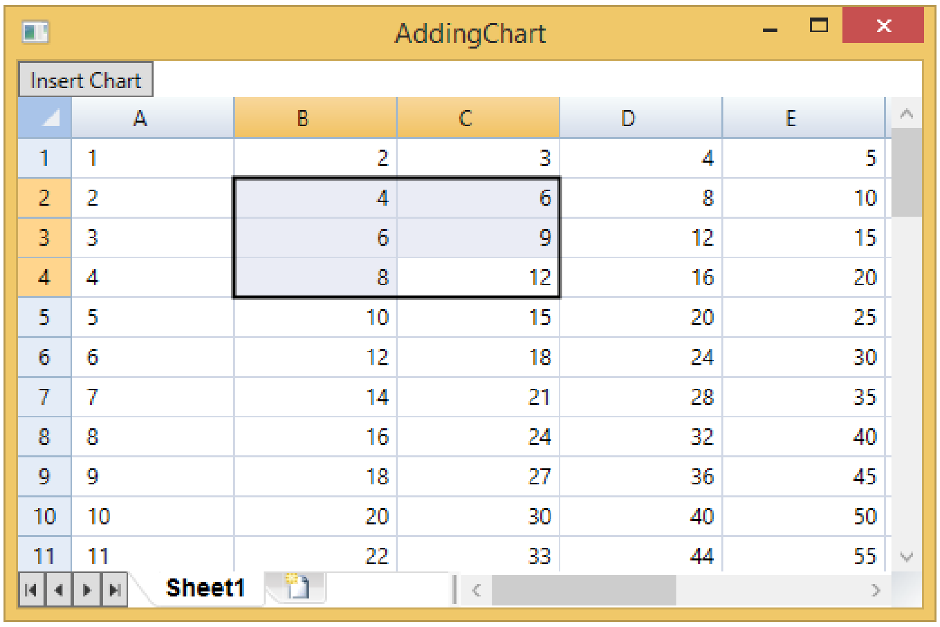
```
If Math.Abs(flex.Selection.RightColumn - flex.Selection.LeftColumn) > 0 AndAlso
   Math.Abs(flex.Selection.BottomRow - flex.Selection.TopRow) > 0 AndAlso
   flex.Selection.IsValid Then
    Dim c1Chart1 As New C1Chart()
    c1Chart1.Data.Children.Clear()
    For row As Integer = flex.Selection.TopRow To flex.Selection.BottomRow
        Dim datas As New List(Of Double)()
        For col As Integer = flex.Selection.LeftColumn To flex.Selection.RightColumn
            Dim value As Object = flex(row, col)
            If value IsNot Nothing AndAlso value.[GetType]().IsNumeric() Then
                Dim cellValue As Double = CDbl(value)
                datas.Add(cellValue)
            End If
        Next
        ' create single series for product price
        Dim ds As New DataSeries()
        'set data
        ds.ValuesSource = datas
        ' add series to the chart
        c1Chart1.Data.Children.Add(ds)
    Next

    ' add item names
    'c1Chart1.Data.ItemNames = ProductNames;
    ' Set chart type
    c1Chart1.ChartType = ChartType.Bar
    flex.InsertChart(c1Chart1)
Else
    MessageBox.Show("Please select more data")
End If
```

- **C#**

```
if (Math.Abs(flex.Selection.RightColumn - flex.Selection.LeftColumn) > 0
  && Math.Abs(flex.Selection.BottomRow - flex.Selection.TopRow) > 0
  && flex.Selection.IsValid)
{
    C1Chart c1Chart1 = new C1Chart();
    c1Chart1.Data.Children.Clear();
    for (int row = flex.Selection.TopRow; row <= flex.Selection.BottomRow; row++)
    {
        List<double> datas = new List<double>();
        for (int col = flex.Selection.LeftColumn; col <= flex.Selection.RightColumn; col++)
        {
            object value = flex[row, col];
            if (value != null && value.GetType().IsNumeric())
```

```
                {
                    double cellValue = (double)value;
                    datas.Add(cellValue);
                }
            }
            // create single series for product price
            DataSeries ds = new DataSeries();
            //set data
            ds.ValuesSource = datas;
            // add series to the chart
            c1Chart1.Data.Children.Add(ds);
        }

        // add item names
        //c1Chart1.Data.ItemNames = ProductNames;
        // Set chart type
        c1Chart1.ChartType = ChartType.Bar;
        flex.InsertChart(c1Chart1);
    }
    else
    {
        MessageBox.Show("Please select more data");
    }
```

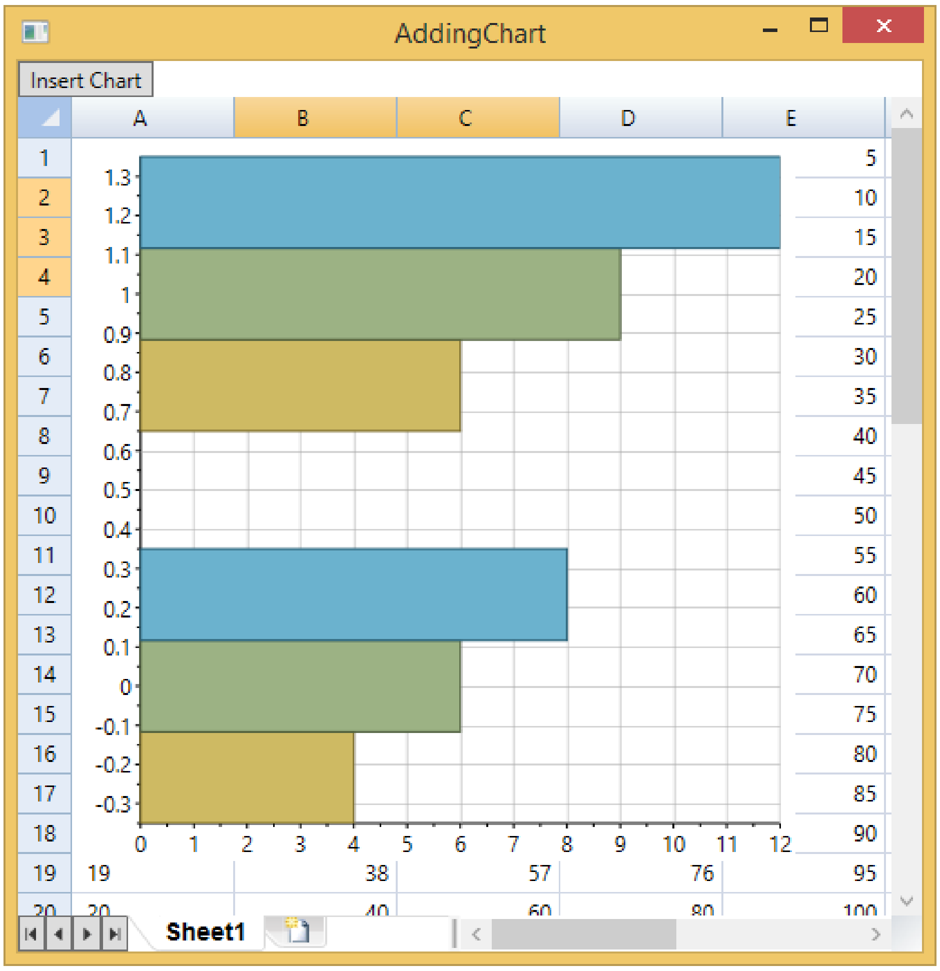
9. Run the application.
10. Select a range of data in cells to display it in form of charts:



	A	B	C	D	E
1	1	2	3	4	5
2	2	4	6	8	10
3	3	6	9	12	15
4	4	8	12	16	20
5	5	10	15	20	25
6	6	12	18	24	30
7	7	14	21	28	35
8	8	16	24	32	40
9	9	18	27	36	45
10	10	20	30	40	50
11	11	22	33	44	55

11. Click **Insert Chart** button.

    The chart displayed on the basis of selected data looks similar to the image given below:



Inserting SparkLine

The data in C1FlexSheet can be visualized using SparkLines. A sparkline is a small graph that fits inside a cell and uses data from a range of cells. To visualize the data in sparklines, you need to add data to a sheet in C1FlexSheet.
Insert the following code directly below the **InitializeComponent()** method to add a sheet and data in it:

- **Visual Basic**

```
flex.AddSheet("Sheet1", 10, 10)
Dim sh = flex.Sheets("Sheet1")

If sh IsNot Nothing Then
    For r As Integer = 0 To 4
        Dim datas As New List(Of Double)()
        For c As Integer = 0 To 5
            If c < 5 Then
                Dim rnd As New Random(New Object().GetHashCode())
                Dim num As Double = Rnd.[Next](10)
                sh.Grid(r, c) = num
                datas.Add(num)
            Else
                'use namespace C1.WPF.FlexGrid for SparkLineType and CellRange
                flex.InsertSparkLine(SparkLineType.Line, datas, sh, New CellRange(r, 5))
            End If
        Next
    Next
End If
```

- **C#**

```
flex.AddSheet("Sheet1", 10, 10);
var sh = flex.Sheets["Sheet1"];

if (sh != null)
{
    for (int r = 0; r < 5; r++)
    {
        List<double> datas = new List<double>();
        for (int c = 0; c < 6; c++)
        {
            if (c < 5)
            {
                Random rnd = new Random(new object().GetHashCode());
                double num = rnd.Next(10);
                sh.Grid[r, c] = num;
                datas.Add(num);
            }
            else
            {
                flex.InsertSparkLine(SparkLineType.Line, datas, sh, new CellRange(r, 5));
                //use namespace C1.WPF.FlexGrid for SparkLineType and CellRange
            }
        }
    }
}
```

The above code will add sparkline to the **C1FlexSheet** control.

Inserting Images

Images can be easily inserted and formatted in C1FlexSheet. Follow the given steps in XAML to add a sheet and insert images in it:

1. Copy the following code inside **Grid** tag to add C1FlexSheet control and a button control:

XAML copyCode

```
    <Grid.RowDefinitions>
        <RowDefinition Height="Auto"/>
        <RowDefinition/>
</Grid.RowDefinitions>
    <c1:C1FlexSheet Name="flex" Grid.Row="1" HorizontalAlignment="Left" VerticalAlignment="Top"/>
    <Button Content="Insert Picture" Name="btnInsertPicture" Click="btnInsertPicture_Click"
HorizontalAlignment="Left" VerticalAlignment="Top" Width="75"/>
```

2. Right-click Design view and select **View Code** from the context menu.
3. Insert the following code directly below the **InitializeComponent()** method to add a sheet:
   - **Visual Basic**

```
'adding Sheet
flex.AddSheet("Sheet1", 50, 10)
```

   - **C#**

```
//adding Sheet
flex.AddSheet("Sheet1", 50, 10);
```

4. Go back to the Design view and select **Event handlers** of **btnInsertPicture** from the **Properties** windows.
5. Double-click the **btnInsertPicture_Click** event handler.
   The Code view will open again.
6. Add the following code to the **btnInsertPicture_Click** event handler to insert an image on button click:
   - **Visual Basic**

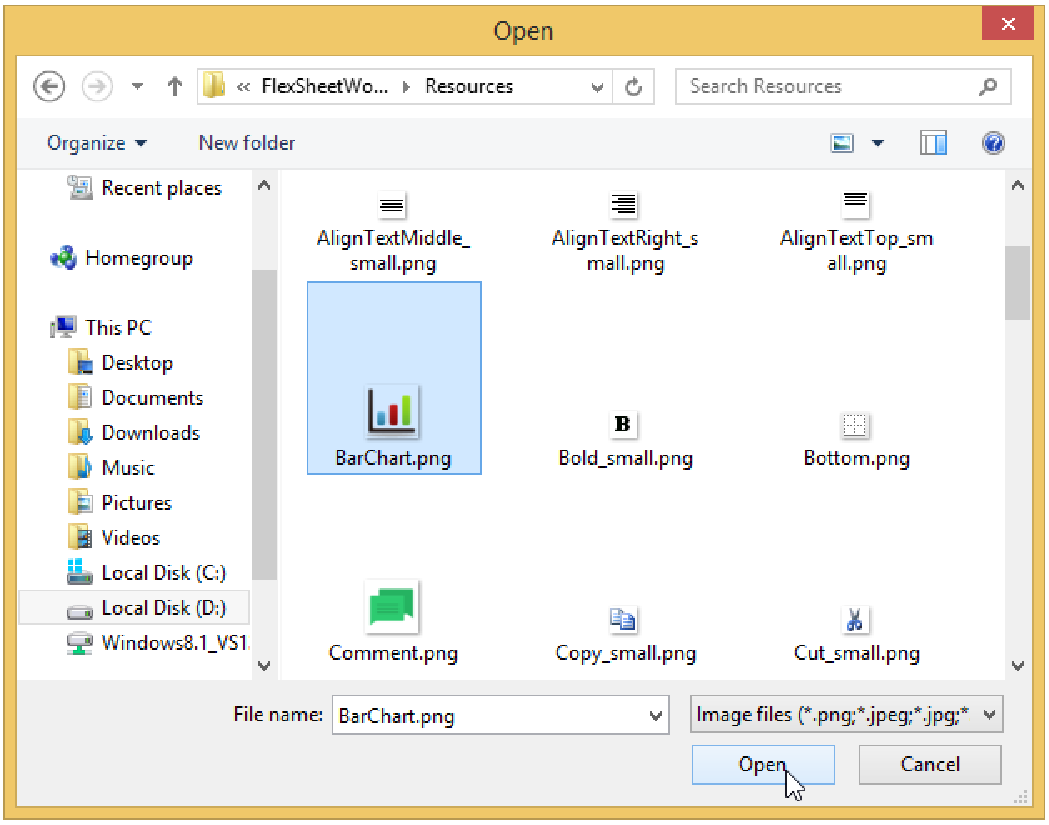
```
Dim dlg = New Microsoft.Win32.OpenFileDialog()
dlg.Filter = "Image files (*.png;*.jpeg;*.jpg;*.bmp)|*.png;*.jpeg;*.jpg;*.bmp|All files (*.*)|*.*"
If dlg.ShowDialog().Value Then
    Try
        Dim b As New BitmapImage()
```

```
            b.BeginInit()
            b.UriSource = New Uri(dlg.FileName)
            b.EndInit()
            flex.InsertImage(b)
        Catch x As Exception
            Dim msg = "Error opening file: " & vbCr & vbLf & vbCr & vbLf + x.Message
            MessageBox.Show(msg, "Error", MessageBoxButton.OK)
        End Try
    End If
```

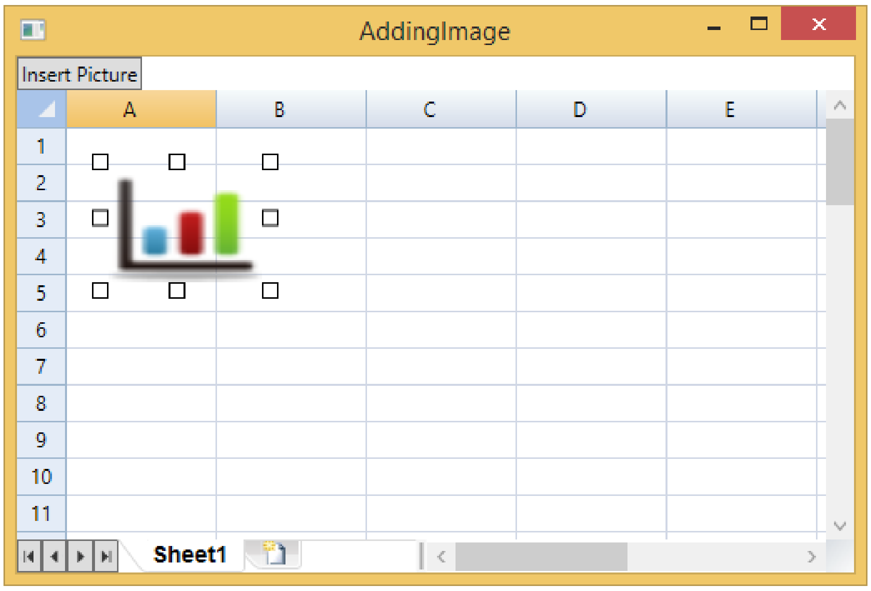
  - **C#**

```
var dlg = new Microsoft.Win32.OpenFileDialog();
dlg.Filter = "Image files (*.png;*.jpeg;*.jpg;*.bmp)|*.png;*.jpeg;*.jpg;*.bmp|All files (*.*)|*.*";
if (dlg.ShowDialog().Value)
{
    try
    {
        BitmapImage b = new BitmapImage();
        b.BeginInit();
        b.UriSource = new Uri(dlg.FileName);
        b.EndInit();
        flex.InsertImage(b);
    }
    catch (Exception x)
    {
        var msg = "Error opening file: \r\n\r\n" + x.Message;
        MessageBox.Show(msg, "Error", MessageBoxButton.OK);
    }
}
```

7. Run the application.
8. Click Insert Picture button.
   The Open dialog box appears.
9. Locate and select an image, and click Open button to open the selected image as shown in the image below:



10. The image appears on the FlexSheet control:

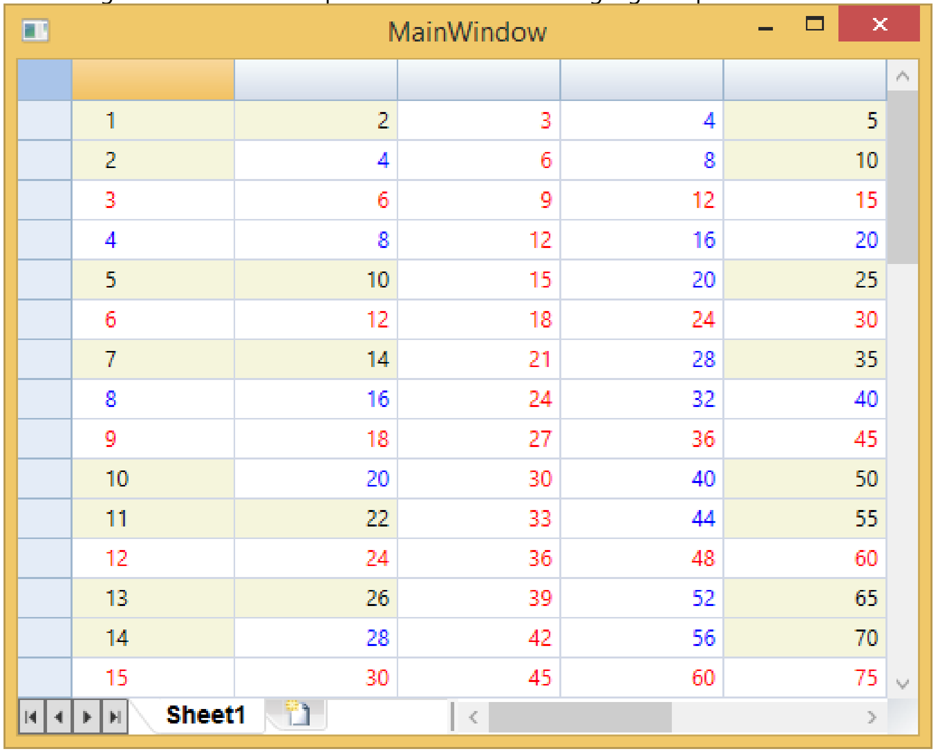


Customizing Cells

C1FlexSheet provides the ability to customize almost every aspect of the appearance of the control, starting from formatting an individual cells to formatting the entire sheet. You will learn about customizing cells in different ways in Applying Conditional Formatting and Cell Formatting.

Applying Conditional Formatting

C1FlexSheet allows you to change the appearance of an individual cell or a range of cells using conditional formatting. Conditional formatting can be used to compare different data and highlight important data.



1	2	3	4	5
2	4	6	8	10
3	6	9	12	15
4	8	12	16	20
5	10	15	20	25
6	12	18	24	30
7	14	21	28	35
8	16	24	32	40
9	18	27	36	45
10	20	30	40	50
11	22	33	44	55
12	24	36	48	60
13	26	39	52	65
14	28	42	56	70
15	30	45	60	75

In the above image, you see that data is formatted with different font color and background color depending on some conditions. The numbers divisible by 3 appear red, numbers divisible by 4 appear blue, and the numbers which are neither divisible by 3 nor 4 appear with beige background color.

The following code illustrates the conditions that we have set to change the appearance of data in cells:

- **Visual Basic**

```
Public Class ConditionalCellFactory
    Inherits CellFactory
    Public Overrides Sub ApplyCellStyles(grid As C1FlexGrid, cellType As CellType,
                                         rng As CellRange, bdr As Border)
        MyBase.ApplyCellStyles(grid, cellType, rng, bdr)

        If cellType = cellType.Cell Then
            Dim tb = If(TypeOf bdr.Child Is TextBlock, TryCast(bdr.Child, TextBlock),
            TryCast(DirectCast(bdr.Child, System.Windows.Controls.Grid).Children(1), TextBlock))

            If grid(rng.Row, rng.Column) IsNot Nothing Then
                If CDbl(grid(rng.Row, rng.Column)) Mod 3 = 0 Then
                    tb.Foreground = New SolidColorBrush(Colors.Red)
                ElseIf CDbl(grid(rng.Row, rng.Column)) Mod 4 = 0 Then
                    tb.Foreground = New SolidColorBrush(Colors.Blue)
                Else
                    bdr.Background = New SolidColorBrush(Colors.Beige)
                End If
            End If
        End If
    End Sub
End Class
```

- **C#**

```
public class ConditionalCellFactory : CellFactory
{
    public override void ApplyCellStyles(C1FlexGrid grid, CellType cellType,
                                         CellRange rng, Border bdr)
    {
        base.ApplyCellStyles(grid, cellType, rng, bdr);

        if (cellType == CellType.Cell)
        {
            var tb = bdr.Child is TextBlock ? bdr.Child as TextBlock : _
            ((System.Windows.Controls.Grid)(bdr.Child)).Children[1] as TextBlock;

            if (grid[rng.Row, rng.Column] != null)
            {
                if ((double)grid[rng.Row, rng.Column] % 3 == 0)
                {
                    tb.Foreground = new SolidColorBrush(Colors.Red);
                }
                else if ((double)grid[rng.Row, rng.Column] % 4 == 0)
                {
                    tb.Foreground = new SolidColorBrush(Colors.Blue);
                }
                else
                {
                    bdr.Background = new SolidColorBrush(Colors.Beige);
                }
            }
        }
    }
}
```

In the above code, a class named **ConditionalCellFactory** is used which inherits from CellFactory class of C1FlexGrid control. The **ConditionalCellFactory** class contains all the implementation of conditional formatting that we have set.

Cell Formatting

C1FlexSheet provides the ability to customize cells and easily set cell styles such as color and font. You can also set different number formats, such as general, number, currency, percentage, and scientific. ExcelCellStyle class is used to set the style and format of cells. ExcelCellStyle class is extended from the FlexGrid CellStyle class that defines attributes used to render grid cells..

Number Formatting

FlexSheet for WPF provides various options to display numbers in different number formats wherein you have the following categories to choose from:

- General
- Number
- Currency
- Percentage
- Scientific

By default, **General** category is selected in C1FlexSheet. **General** category does not have any special rule of formatting. You can change the number formatting of the cells to number, decimal, currency, percentage or scientific format. For example, to create a worksheet for your monthly or yearly business budgets, you can show the monetary values in **Currency** number format.

To format numbers in your C1FlexSheet worksheet, follow the steps given below:

1. Add **C1FlexSheet** control and **C1Toolbar** to your application.
2. Add a ComboBox, NumericBox and a button to the ToolBar using the following code:

XAML copyCode

```
<ComboBox x:Name="formatList" ItemsSource="{Binding NumberFormats}" Width="80"
          DisplayMemberPath="Name" Height="25"
          SelectedValue="{Binding SelectedFormat, Mode=TwoWay}"
          SelectionChanged="formatList_SelectionChanged"></ComboBox>
<c1:C1NumericBox x:Name="decimalPlacesBox" Minimum="0" Margin="10,0,0,0"
ValueChanged="decimalPlacesBox_ValueChanged"></c1:C1NumericBox>
<Button x:Name="numberButton" Content="Apply" Click="numberButton_Click"
Margin="20,10,0,0"></Button>
```

3. Create a class named NumberFormat which inherits **INotifyPropertyChanged** interface. **NumberFormat** class contains SelectedFormat property to get and set the selected format of numbers.
4. Add the following code in Code view to set the SelectedFormat property of **NumberFormat** class:

Visual Basic

```
Public Property SelectedFormat() As NumberFormat
        Get
                Return _selectedFormat
        End Get
        Set
                _selectedFormat = value
        End Set
End Property
```

   - **C#**

```
public NumberFormat SelectedFormat
{
    get
    {
        return _selectedFormat;
    }
    set
    {
        _selectedFormat = value;
    }
}
```

Also, add the following code in Code view to set the **NumberFormats** property of **ObservableCollection** data collection of Number type:

Visual Basic

```
Public Property NumberFormats() As ObservableCollection(Of
NumberFormat)
        Get
                Return _numberFormats
        End Get
        Set
                _numberFormats = value
        End Set
End Property
```

- **C#**

```
public ObservableCollection<NumberFormat> NumberFormats
{
    get
    {
        return _numberFormats;
    }
    set
    {
        _numberFormats = value;
    }
}
```

5. Define **InitializeNumberFormats()** method to specify the formats that will be used to change the number formatting:

Visual Basic

```
Public Sub InitializeNumberFormats()
        NumberFormats = New ObservableCollection(Of NumberFormat)()

        Dim generalFormat As New NumberFormat()
        generalFormat.Name = "General"
        generalFormat.Format = "G"
        NumberFormats.Add(generalFormat)

        Dim numberFormat As New NumberFormat()
        numberFormat.Name = "Number"
        numberFormat.Format = "N"
        NumberFormats.Add(numberFormat)

        Dim currencyFormat As New NumberFormat()
        currencyFormat.Name = "Currency"
        currencyFormat.Format = "C"
        NumberFormats.Add(currencyFormat)

        Dim percentFormat As New NumberFormat()
        percentFormat.Name = "Percentage"
        percentFormat.Format = "P"
        NumberFormats.Add(percentFormat)
        Dim exponentialFormat As New NumberFormat()
        exponentialFormat.Name = "Scientific"
        exponentialFormat.Format = "E"
        NumberFormats.Add(exponentialFormat)

        SelectedFormat = generalFormat
End Sub
```

- **C#**

```
public void InitializeNumberFormats()
{
    NumberFormats = new ObservableCollection<NumberFormat>();

    NumberFormat generalFormat = new NumberFormat();
    generalFormat.Name = "General";
    generalFormat.Format = "G";
    NumberFormats.Add(generalFormat);

    NumberFormat numberFormat = new NumberFormat();
    numberFormat.Name = "Number";
    numberFormat.Format = "N";
    NumberFormats.Add(numberFormat);

    NumberFormat currencyFormat = new NumberFormat();
    currencyFormat.Name = "Currency";
    currencyFormat.Format = "C";
    NumberFormats.Add(currencyFormat);

    NumberFormat percentFormat = new NumberFormat();
    percentFormat.Name = "Percentage";
    percentFormat.Format = "P";
    NumberFormats.Add(percentFormat);

    NumberFormat exponentialFormat = new NumberFormat();
    exponentialFormat.Name = "Scientific";
    exponentialFormat.Format = "E";
    NumberFormats.Add(exponentialFormat);

    SelectedFormat = generalFormat;
}
```

6. Add the following code to the FormatList ComboBox that monitors the selection and sends information to the console to show the list of available number formats when the selection changes:

Visual Basic

```
Dim selectedFormat__1 = TryCast(e.AddedItems(0), NumberFormat)

Select Case selectedFormat__1.Name
        Case "General"
                decimalPlacesBox.Visibility =
System.Windows.Visibility.Collapsed
                decimalPlacesBox.Value = 0
                SelectedFormat.Format = selectedFormat__1.Format
                Exit Select
        Case "Number"
                decimalPlacesBox.Visibility =
System.Windows.Visibility.Visible
                SelectedFormat.Format = selectedFormat__1.Format
                Exit Select
        Case "Currency"
                decimalPlacesBox.Visibility =
System.Windows.Visibility.Visible
                SelectedFormat.Format = selectedFormat__1.Format
                Exit Select
        Case "Percentage"
                decimalPlacesBox.Visibility =
System.Windows.Visibility.Visible
                SelectedFormat.Format = selectedFormat__1.Format
                Exit Select
        Case "Scientific"
                decimalPlacesBox.Visibility =
System.Windows.Visibility.Visible
                SelectedFormat.Format = selectedFormat__1.Format
                Exit Select
End Select
```

- **C#**

```
var selectedFormat = e.AddedItems[0] as NumberFormat;

switch (selectedFormat.Name)
{
    case "General":
        decimalPlacesBox.Visibility = System.Windows.Visibility.Collapsed;
        decimalPlacesBox.Value = 0;
        SelectedFormat.Format = selectedFormat.Format;
        break;
    case "Number":
        decimalPlacesBox.Visibility = System.Windows.Visibility.Visible;
        SelectedFormat.Format = selectedFormat.Format;
        break;
    case "Currency":
        decimalPlacesBox.Visibility = System.Windows.Visibility.Visible;
        SelectedFormat.Format = selectedFormat.Format;
        break;
    case "Percentage":
        decimalPlacesBox.Visibility = System.Windows.Visibility.Visible;
        SelectedFormat.Format = selectedFormat.Format;
        break;
    case "Scientific":
        decimalPlacesBox.Visibility = System.Windows.Visibility.Visible;
        SelectedFormat.Format = selectedFormat.Format;
        break;
}
```

7. Add the given code to avail the selection of decimal places while changing the number format:

Visual Basic

```
SelectedFormat.DecimalPlaces = CInt(e.NewValue)
```

- **C#**

```
SelectedFormat.DecimalPlaces = (int)e.NewValue;
```

8. Add this code on the click event of the button, numberButton in our case, to apply the selected format in the previous step:

Visual Basic

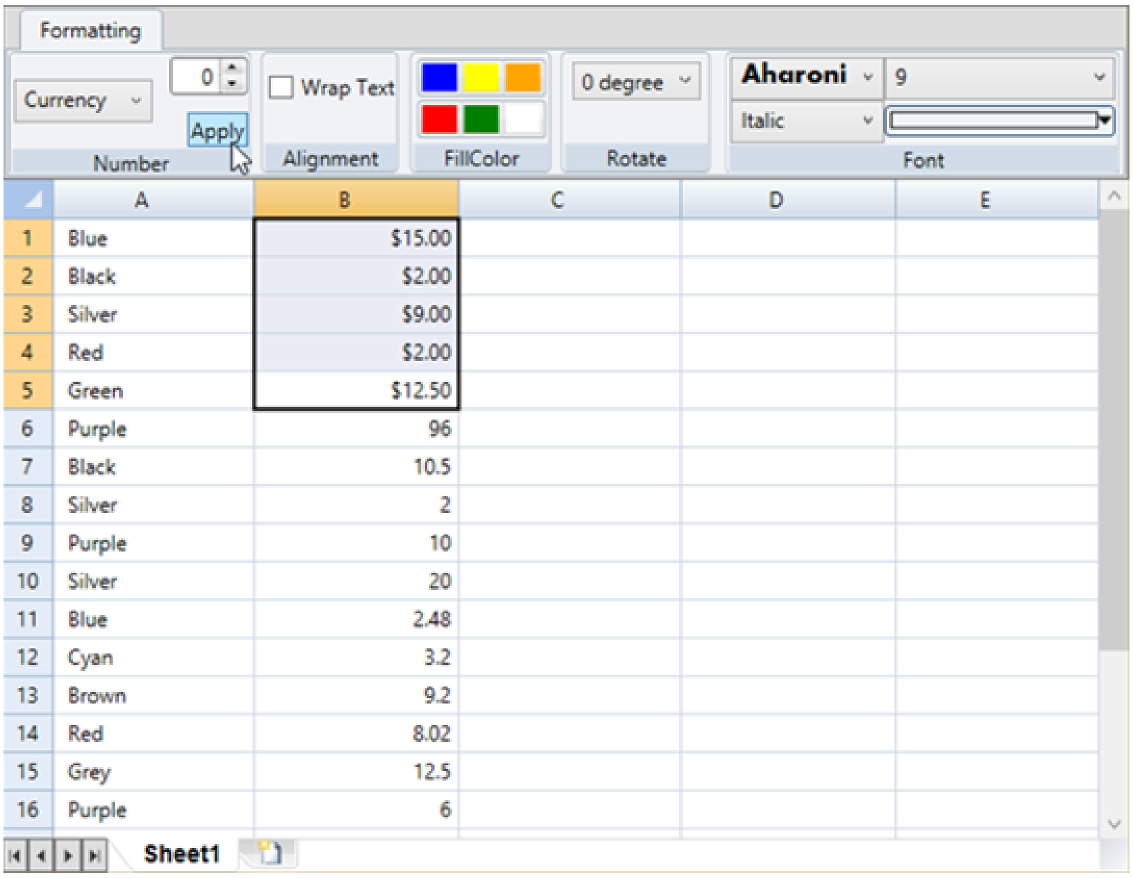
```
Dim cellrange = flex.Selection.Cells
```

```
For Each rng As var In cellrange
        If rng.IsValid Then
                Dim row = TryCast(flex.Rows(rng.Row), ExcelRow)
                Dim col = flex.Columns(rng.Column)

                Dim excelCellStyle As New ExcelCellStyle()

                If row IsNot Nothing Then
                        Dim cs = TryCast(row.GetCellStyle(col),
ExcelCellStyle)
                        If cs IsNot Nothing Then
                                excelCellStyle = cs
                        End If
                End If

                'set selected Number formatting on cells
                excelCellStyle.Format = SelectedFormat.Format +
SelectedFormat.DecimalPlaces
                row.SetCellStyle(col, excelCellStyle)

                flex.Invalidate(rng)
        End If
Next
```

- **C#**

```
var cellrange = flex.Selection.Cells;

foreach (var rng in cellrange)
{
    if (rng.IsValid)
    {
        var row = flex.Rows[rng.Row] as ExcelRow;
        var col = flex.Columns[rng.Column];

        ExcelCellStyle excelCellStyle = new ExcelCellStyle();

        if (row != null)
        {
            var cs = row.GetCellStyle(col) as ExcelCellStyle;
            if (cs != null)
                excelCellStyle = cs;
        }

        //set selected Number formatting on cells
        excelCellStyle.Format = SelectedFormat.Format + SelectedFormat.DecimalPlaces;
        row.SetCellStyle(col, excelCellStyle);

        flex.Invalidate(rng);
    }
}
```

9. Select the cells with numeric data to change the number format.
10. Select a category from the available category list in Number tab.
    Set the decimal places if required.
11. Click Apply button to see the applied number format.
    On applying number format, FlexSheet control looks similar to the image given below:



	A	B	C	D	E
1	Blue	$15.00			
2	Black	$2.00			
3	Silver	$9.00			
4	Red	$2.00			
5	Green	$12.50			
6	Purple	96			
7	Black	10.5			
8	Silver	2			
9	Purple	10			
10	Silver	20			
11	Blue	2.48			
12	Cyan	3.2			
13	Brown	9.2			
14	Red	8.02			
15	Grey	12.5			
16	Purple	6			

In the above image, the selected number format is Currency.

Setting Cell Alignment and Indentation

FlexSheet for WPF allows you to align and indent cells to enhance the appearance of the presented data. Cell alignment can be easily applied using SetCellFormat method and indentation can be applied using SetCellIndent method in C1FlexSheet.

You can change the alignment of data both horizontally and vertically, and apply indentation to the cell contents. Follow the given steps to apply alignment and indentation in cells:

1. Add a C1FlexSheet control to the **MainWindow.xaml**.
2. Add the following code in XAML to insert the buttons which will perform cell alignment and indentation:

XAML copyCode

```
<Button Content="Left Align" Name="_btnLeft" Click="_btnLeft_Click"
HorizontalAlignment="Left" VerticalAlignment="Top" Width="60"/>
<Button Content="Center Align" Name="_btnCenter" Click="_btnCenter_Click"
HorizontalAlignment="Left" VerticalAlignment="Top" Width="85"
Margin="78,0,0,0"/>
<Button Content="Right Align" Name="_btnRight" Click="_btnRight_Click"
HorizontalAlignment="Left" VerticalAlignment="Top" Width="70"
Margin="185,0,0,0"/>
<Button Content="Top Align" Name="_btnTop" Click="_btnTop_Click"
HorizontalAlignment="Left" VerticalAlignment="Top" Width="65"
Margin="22,25,0,0"/>
<Button Content="Middle Align" Name="_btnMiddle" Click="_btnMiddle_Click"
```

```
HorizontalAlignment="Left" VerticalAlignment="Top" Width="80"
Margin="107,25,0,0"/>
<Button Content="Bottom Align" Name="_btnBottom" Click="_btnBottom_Click"
HorizontalAlignment="Left" VerticalAlignment="Top" Width="80"
Margin="212,25,0,0"/>
<Button Content="Decrease Indent" Name="_btnDecreaseIndent"
Click="_btnDecreaseIndent_Click" HorizontalAlignment="Right"
VerticalAlignment="Top" Width="100" Margin="0,25,10,0"/>
<Button Content="Increase Indent" Name="_btnIncreaseIndent"
Click="_btnIncreaseIndent_Click" HorizontalAlignment="Right"
VerticalAlignment="Top" Width="100" Margin="0,0,10,0"/>
```

3. Right-click Design view and select **View Code** from the context menu.
4. Insert the following code directly below the **InitializeComponent()** method to add a sheet in FlexSheet control and data in the sheet:
   - **Visual Basic**

```
flex.AddSheet("Sheet 1", 12, 6)
Dim sheet = flex.Sheets("Sheet 1")
sheet.Grid.Columns(1).DataType = GetType(Decimal)

sheet.Grid(0, 0) = "Blue"
sheet.Grid(1, 0) = "Black"
sheet.Grid(2, 0) = "Silver"
sheet.Grid(3, 0) = "apple"
sheet.Grid(4, 0) = "Green"
sheet.Grid(5, 0) = "Purple"
sheet.Grid(6, 0) = "Black"
sheet.Grid(7, 0) = "Silver"
sheet.Grid(8, 0) = "Black"
sheet.Grid(9, 0) = "Silver"

sheet.Grid(0, 1) = 15
sheet.Grid(1, 1) = 2
sheet.Grid(2, 1) = 9.1
sheet.Grid(3, 1) = 2
sheet.Grid(4, 1) = 29.89
sheet.Grid(5, 1) = 93.6
sheet.Grid(6, 1) = 0.1
sheet.Grid(7, 1) = 2
```

   - **C#**

```
flex.AddSheet("Sheet 1", 12, 6);
var sheet = flex.Sheets["Sheet 1"];
sheet.Grid.Columns[1].DataType = typeof(decimal);

sheet.Grid[0, 0] = "Blue";
sheet.Grid[1, 0] = "Black";
sheet.Grid[2, 0] = "Silver";
sheet.Grid[3, 0] = "apple";
sheet.Grid[4, 0] = "Green";
sheet.Grid[5, 0] = "Purple";
sheet.Grid[6, 0] = "Black";
sheet.Grid[7, 0] = "Silver";
sheet.Grid[8, 0] = "Black";
sheet.Grid[9, 0] = "Silver";

sheet.Grid[0, 1] = 15;
sheet.Grid[1, 1] = 2;
sheet.Grid[2, 1] = 9.1;
```

```
    sheet.Grid[3, 1] = 2;
    sheet.Grid[4, 1] = 29.89;
    sheet.Grid[5, 1] = 93.6;
    sheet.Grid[6, 1] = 0.1;
    sheet.Grid[7, 1] = 2;
    sheet.Grid[8, 1] = 10;
```

After adding data to a sheet in **C1FlexSheet** control, you can choose how to display the data in cells using different types of indentation and alignment.

Horizontal Alignment

You can change the horizontal alignment of the data in cell by applying left, centre or right alignment to it. Just select the cell(s) to align the data and you are good to go.

To align the data on the left, use the following code:

- **Visual Basic**

```
flex.SetCellFormat(flex.Selection.Cells, CellFormat.HorizontalAlignment,
                   HorizontalAlignment.Left)
```

- **C#**

```
flex.SetCellFormat(flex.Selection.Cells, CellFormat.HorizontalAlignment,
                   HorizontalAlignment.Left);
```

Similarly, you can align the cell data on the right or at the center by changing the value of enum **HorizontalAlignment** from **Left** to **Right** or **Center**.

Vertical Alignment

You can change the vertical alignment of the data in cell by applying top, middle or right alignment to it. Just select the cell(s) to align the data vertically.

To align the data at the top, use the following code:

- **Visual Basic**

```
flex.SetCellFormat(flex.Selection.Cells, CellFormat.VerticalAlignment,
                   VerticalAlignment.Top)
```

- **C#**

```
flex.SetCellFormat(flex.Selection.Cells, CellFormat.VerticalAlignment,
                   VerticalAlignment.Top);
```

Similarly, you can align the data in the middle or at the bottom of the cell by changing the value of enum **VerticalAlignment** from **Top** to **Middle** or **Bottom**.

Indentation

You can change the indentation of a cell by increasing or decreasing the indent of the selected cell(s). To increase the cell indention, use the following code:

- **Visual Basic**

```
_currentIndent += 12
flex.SetCellIndent(flex.Selection.Cells, _currentIndent)
```

- **C#**

```
_currentIndent += 12;
flex.SetCellIndent(flex.Selection.Cells, _currentIndent);
```

To decrease the cell indention, use the following code:

- **Visual Basic**

```
_currentIndent -= 12
If _currentIndent >= 0 Then
    flex.SetCellIndent(flex.Selection.Cells, _currentIndent)
Else
    _currentIndent = 0
End If
```

- **C#**

```
_currentIndent -= 12;
if (_currentIndent >= 0)
    flex.SetCellIndent(flex.Selection.Cells, _currentIndent);
else
    _currentIndent = 0;
```

Wrap Text

Text wrapping can be done when there is too much data to be displayed in a single cell. Wrapping cell data gives you an advantage to display large amount of data in multiple lines in a single cell.

You might find text wrapping useful where you need to write long strings in a cell. For example, you need to create a list of vendors with all the details regarding products they supply, including their address details. Address can be lengthy so wrapping text can help in writing multiple lines of address in a single cell. CellFormat.TextWrappping enum can be used to wrap the text.

Perform the given steps to wrap text in a cell in C1FlexSheet:

1. Add a CheckBox in your application to wrap text in a cell.
2. Add the following code to the **Checked** event of the CheckBox:

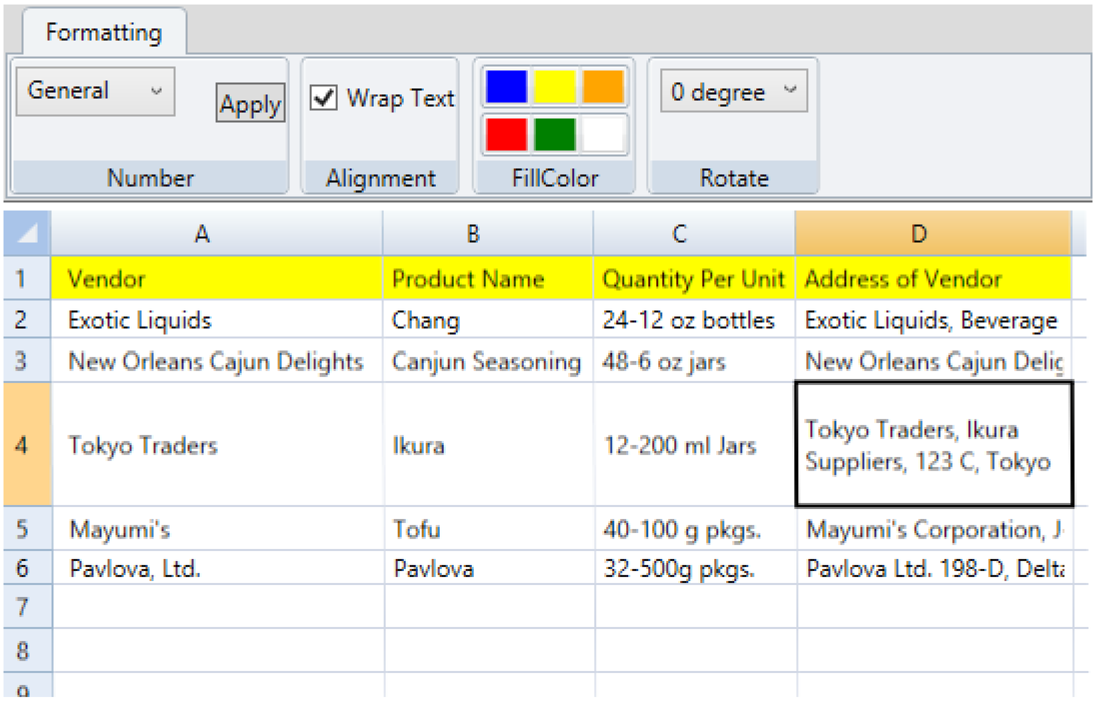
   Visual Basic

   ```
   Dim cellRange = flex.Selection.Cells
   flex.SetCellFormat(cellRange, CellFormat.TextWrapping,
   chkWrapText.IsChecked)

   flex.Invalidate()
   ```

   - **C#**

   ```
   var cellRange = flex.Selection.Cells;
   flex.SetCellFormat(cellRange, CellFormat.TextWrapping, chkWrapText.IsChecked);

   flex.Invalidate();
   ```

   This code wraps the text once the Wrap Text checkbox is checked and the output looks similar to the following image:



	A	B	C	D
1	Vendor	Product Name	Quantity Per Unit	Address of Vendor
2	Exotic Liquids	Chang	24-12 oz bottles	Exotic Liquids, Beverage
3	New Orleans Cajun Delights	Canjun Seasoning	48-6 oz jars	New Orleans Cajun Delic
4	Tokyo Traders	Ikura	12-200 ml Jars	Tokyo Traders, Ikura Suppliers, 123 C, Tokyo
5	Mayumi's	Tofu	40-100 g pkgs.	Mayumi's Corporation, J
6	Pavlova, Ltd.	Pavlova	32-500g pkgs.	Pavlova Ltd. 198-D, Delta
7				
8				

To unwrap text in a cell, click Wrap Text check box again. Add the following code to the **Unchecked** event of the CheckBox:

Visual Basic

```
Dim cellRange = flex.Selection.Cells
flex.SetCellFormat(cellRange, CellFormat.TextWrapping,
chkWrapText.IsChecked)
flex.Invalidate()
```

- **C#**

```
var cellRange = flex.Selection.Cells;
flex.SetCellFormat(cellRange, CellFormat.TextWrapping, chkWrapText.IsChecked);
flex.Invalidate();
```

Merge Cells

FlexSheet for WPF allows you to merge cells in a situation where you need to combine the data in multiple cells into a single cell. For example, you want to create a project timeline worksheet where you can track your project deadlines and status, and want to give it a title heading as well. To create a title heading, you need to combine multiple cells into one, which can be achieved by merging cells.

Merging in C1FlexSheet can be performed using ExcelMergeManager class. The code given below is used to merge cells in **C1FlexSheet**. In our case, we have implemented this code on the click event of a button.

- **Visual Basic**

```
' get current selection, ensure there's more than one cell in it
Dim sel = flex.Selection
Dim xmm = TryCast(flex.MergeManager, ExcelMergeManager)
If xmm IsNot Nothing Then
    ' check if the selection contains any merged ranges
    Dim hasMerges = False
    For Each cell As CellRange In sel.Cells
        If Not xmm.GetMergedRange(flex, CellType.Cell, cell).IsSingleCell Then
```

```
            hasMerges = True
        End If
    Next
    ' toggle merging for the selection
    If hasMerges Then
        ' clear merged ranges
        xmm.RemoveRange(sel)
    Else
        ' merge selection
        xmm.AddRange(sel)
    End If
    ' show changes
    flex.Invalidate()
End If
```

- **C#**

```
// get current selection, ensure there's more than one cell in it
var sel = flex.Selection;
var xmm = flex.MergeManager as ExcelMergeManager;
if (xmm != null)
{
    // check if the selection contains any merged ranges
    var hasMerges = false;
    foreach (var cell in sel.Cells)
    {
        if (!xmm.GetMergedRange(flex, CellType.Cell, cell).IsSingleCell)
        {
            hasMerges = true;
        }
    }
    // toggle merging for the selection
    if (hasMerges)
    {
        // clear merged ranges
        xmm.RemoveRange(sel);
    }
    else
    {
        // merge selection
        xmm.AddRange(sel);
    }
    // show changes
    flex.Invalidate();
}
```

Formatting Font

You might want to change the appearance of the text in cell(s) so that the text stands out from the rest. In C1FlexSheet, you can easily format the font of text in a cell by changing the font style, font family, font size, and font color.

SetCellFormat method is used to set the format for the cell in C1FlexSheet.

The following code uses SetCellFormat method to change the font family of the font in a cell:

Visual Basic

```
If flex IsNot Nothing Then
    flex.SetCellFormat(flex.Selection.Cells, CellFormat.FontFamily,
                       fontFamilyCombo.SelectedValue)
End If
```

- **C#**

```
if (flex != null)
    flex.SetCellFormat(flex.Selection.Cells, CellFormat.FontFamily,
                       fontFamilyCombo.SelectedValue);
```

To change the font size, first you need to set the size limit of the font. The following code uses SetCellFormat method to change the font size after setting the limit:

Visual Basic

```
If flex IsNot Nothing Then
    flex.SetCellFormat(flex.Selection.Cells, CellFormat.FontSize,
                       fontSize.SelectedValue)
End If
```

- **C#**

```
if (flex != null)
    flex.SetCellFormat(flex.Selection.Cells, CellFormat.FontSize,
                       fontSize.SelectedValue);
```

The following code illustrates the use of **SetCellFormat** method to change the Font style:

Visual Basic

```
If flex IsNot Nothing Then
    Select Case fontStyle.SelectedValue.ToString()
        Case "Bold"
            flex.SetCellFormat(flex.Selection.Cells, CellFormat.FontWeight,
FontWeights.Bold)
            Exit Select
        Case "Italic"
            flex.SetCellFormat(flex.Selection.Cells, CellFormat.FontStyle,
FontStyles.Italic)
            Exit Select
        Case "BoldAndItalic"
            flex.SetCellFormat(flex.Selection.Cells, CellFormat.FontWeight,
FontWeights.Bold)
            flex.SetCellFormat(flex.Selection.Cells, CellFormat.FontStyle,
FontStyles.Italic)
            Exit Select
        Case Else
            flex.SetCellFormat(flex.Selection.Cells, CellFormat.FontStyle,
FontStyles.Normal)
            flex.SetCellFormat(flex.Selection.Cells, CellFormat.FontWeight,
FontWeights.Normal)
            Exit Select
    End Select
End If
```

- **C#**

```
if (flex != null)
{
    switch (fontStyle.SelectedValue.ToString())
    {
        case "Bold": flex.SetCellFormat(flex.Selection.Cells, CellFormat.FontWeight,
                                        FontWeights.Bold);
            break;
        case "Italic": flex.SetCellFormat(flex.Selection.Cells, CellFormat.FontStyle,
                                          FontStyles.Italic);
            break;
        case "BoldAndItalic": flex.SetCellFormat(flex.Selection.Cells, CellFormat.FontWeight,
                                                               FontWeights.Bold);
            flex.SetCellFormat(flex.Selection.Cells, CellFormat.FontStyle, FontStyles.Italic);
            break;
        default: flex.SetCellFormat(flex.Selection.Cells, CellFormat.FontStyle, FontStyles.Normal);
            flex.SetCellFormat(flex.Selection.Cells, CellFormat.FontWeight, FontWeights.Normal);
            break;
    }
}
```

You can create your own color picker using system colors and use its reference in the code to change the font color. The following code can then be used to change the font color:

Visual Basic

```
If flex IsNot Nothing Then
   flex.SetCellFormat(flex.Selection.Cells, CellFormat.Foreground,
                      New SolidColorBrush(obj))
End If
```

- **C#**

```
if (flex != null)
    flex.SetCellFormat(flex.Selection.Cells, CellFormat.Foreground, new SolidColorBrush(obj));
```

Fill Color

C1FlexSheet allows you to fill colors in cells to highlight particular data. The data can be highlighted by filling back color in cells. For example, to fill orange color in a cell, you can use the following code:

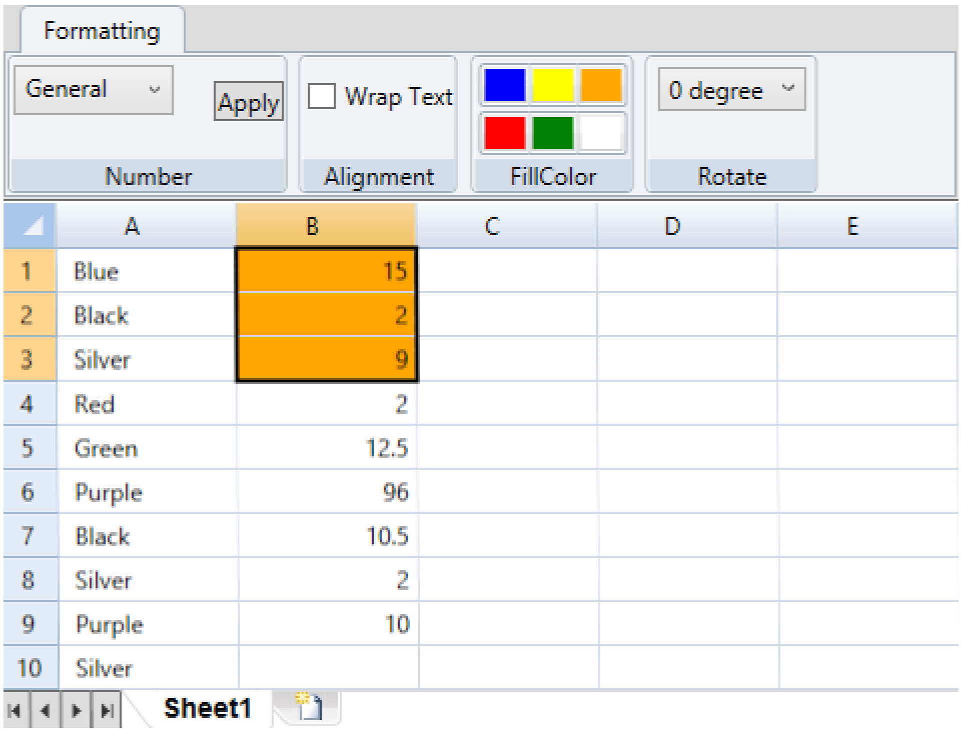
Visual Basic

```
flex.SetCellFormat(flex.Selection.Cells, CellFormat.Background, New
SolidColorBrush(Colors.Orange))
```

- **C#**

```
flex.SetCellFormat(flex.Selection.Cells, CellFormat.Background,
                   new SolidColorBrush(Colors.Orange));
```

After filling color, the selected cell looks similar to the following:



	A	B	C	D	E
1	Blue	15			
2	Black	2			
3	Silver	9			
4	Red	2			
5	Green	12.5			
6	Purple	96			
7	Black	10.5			
8	Silver	2			
9	Purple	10			
10	Silver				

Similarly, you can add and apply any color of your choice, available in the predefined set of system colors, to the selected cells at runtime.

You can also remove the fill color applied to a cell by adding the following code:

Visual Basic

```
flex.SetCellFormat(flex.Selection.Cells, CellFormat.Background, Nothing)
```

- **C#**

```
flex.SetCellFormat(flex.Selection.Cells, CellFormat.Background, null);
```

Rotating Text

You might want to display text in a cell at a particular angle for optimal display. You can rotate the text at different angles in C1FlexSheet to display text at a certain angle. To rotate text in a cell, use C1FlexSheet.SetCellAngle method.

To provide option to rotate cell data at a certain angle, follow the given steps:

1. Add the following code to your application to add a ComboBox and set its properties:

XAML copyCode

```
<ComboBox x:Name="cmbRotate" SelectedIndex="0" Margin="3"
      ToolTipService.ToolTip="Text Rotate"
SelectionChanged="cmbRotate_SelectionChanged">
    <ComboBox.Items>
        <ComboBoxItem Content="0 degree" />
        <ComboBoxItem Content="45 degree" />
        <ComboBoxItem Content="90 degree" />
        <ComboBoxItem Content="135 degree" />
        <ComboBoxItem Content="180 degree" />
    </ComboBox.Items>
</ComboBox>
```

2. Add the following code to the **SelectedChanged** event of the ComboBox to rotate text of the selected cells using SetCellAngle method:

Visual Basic

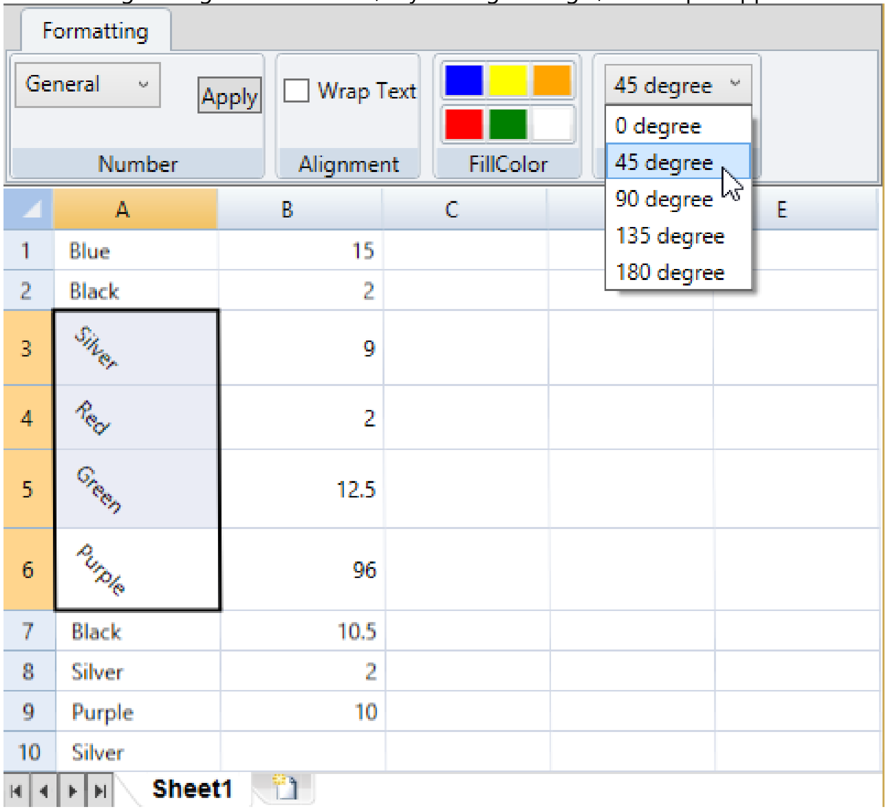
```
Dim index = DirectCast(sender, ComboBox).SelectedIndex
Dim angle As Double = 0
Select Case index
        Case 0
                angle = 0
                Exit Select
        Case 1
                angle = 45
                Exit Select
        Case 2
                angle = 90
                Exit Select
        Case 3
                angle = 135
                Exit Select
        Case 4
                angle = 180
                Exit Select
        Case Else
                angle = 0
                Exit Select
End Select
If flex IsNot Nothing Then
        flex.SetCellAngle(flex.Selection.Cells, angle, 8)
End If
```

- **C#**

```
var index = ((ComboBox)sender).SelectedIndex;
double angle = 0;
switch (index)
{
    case 0:
        angle = 0;
        break;
    case 1:
        angle = 45;
        break;
    case 2:
        angle = 90;
        break;
    case 3:
        angle = 135;
        break;
    case 4:
        angle = 180;
        break;
    default:
        angle = 0;
        break;
}
if (flex != null)
    flex.SetCellAngle(flex.Selection.Cells, angle, 8);
```

The given code allows you to rotate the cell data at 45, 90, 135, and 180 degree angles. You can customize the code and add the angle at which you want your data to be displayed.

On selecting an angle to rotate text, say 45 degree angle, the output appears similar to the following:



	A	B	C	D	E
1	Blue	15			
2	Black	2			
3	Silver	9			
4	Red	2			
5	Green	12.5			
6	Purple	96			
7	Black	10.5			
8	Silver	2			
9	Purple	10			
10	Silver				

Customizing Edit Options

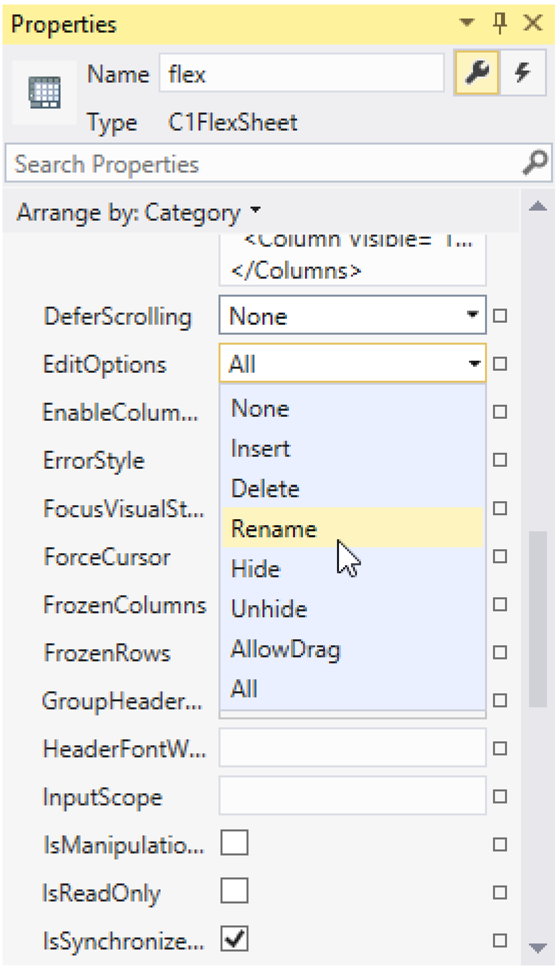
FlexSheet for WPF provides various edit options on a multi-tabbed spreadsheet control. The edit options are as follows:

- **None:** Does not show context menu of FlexSheet tabs
- **Insert:** Shows context menu that inserts a new sheet
- **Delete:** Shows context menu that deletes a selected sheet
- **Rename:** Shows context menu that renames a selected sheet
- **Hide:** Shows context menu that hide a selected sheet
- **Unhide:** Shows context menu that unhide all the hidden sheet
- **AllowDrag:** Allows to drag the sheet tabs
- **All:** Shows all edit options in context menu and allows to drag the sheet tabs

These edit options can be set using EditOptions property and are displayed when you right-click any sheet tab at run time. EditOptions property controls which context menu should be shown when you right-click on the tab strip.

In Design View

1. In Design view, select the FlexSheet control.
2. Navigate to the **Properties window** and locate the **EditOptions** property in **Miscellaneous** drop-down section.
3. Click the **EditOptions** drop-down menu and select the edit options you want to add in the context menu. By default, the **EditOptions** property is set to **All**. Here, we have selected **Rename** to only display rename option in the context menu.


Properties
Name flex
Type C1FlexSheet
Search Properties
Arrange by: Category
<Column Visible= T...
</Columns>
DeferScrolling None
EditOptions All
EnableColum...
ErrorStyle
FocusVisualSt...
ForceCursor
FrozenColumns
FrozenRows
GroupHeader...
HeaderFontW...
InputScope
IsManipulatio... ☐
IsReadOnly ☐
IsSynchronize... ✓

As you can see in the image, EditOptions has 8 values, None, Insert, Delete, Rename, Hide, Unhide, AllowDrag and All. These values are specified in FlexSheetEditOptions enum that defines the edit options.

> **Note:** The AllowDrag property does not appear in the context menu but once applied it allows you to drag or move sheets in tab strip.

In XAML

You can also customize the **EditOptions** appearing in the context menu by setting the EditOptions property in XAML View using the following code:

XAML copyCode

```
<c1:C1FlexSheet x:Name="flexsheet1" BorderBrush="Gray" BorderThickness="1"
                Grid.Row="1" Width="1000"  HorizontalAlignment="Left"
ShowSingleTab="False" EditOptions="Rename" />
```

In Code

You can choose to show the EditOptions context menu options using the following code:

- **Visual Basic**

```
flexsheet1.EditOptions = FlexSheetEditOptions.Rename
```

- **C#**

```
flexsheet1.EditOptions = FlexSheetEditOptions.Rename;
```

Customizing Tab Strip

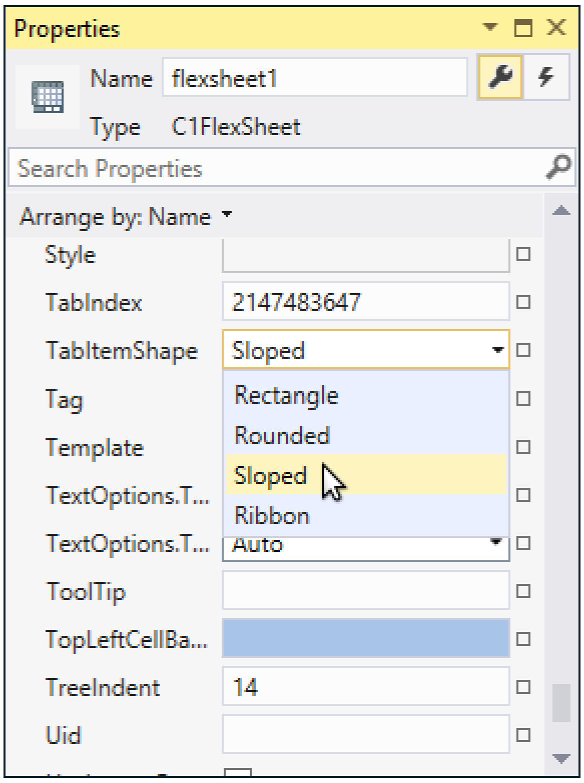
A Tab strip can be viewed as soon as a new sheet is added to C1FlexSheet. Tab strip is used to show multiple sheets in the same area in a C1FlexSheet control. You can customize the tab strip according to your need. To learn about tab strip customization, see Customizing Tab Shape and Hiding Tab Strip.

Customizing Tab Shape

FlexSheet for WPF allows you to set different geometric shapes of the sheet tabs appearing in the Tab Strip of a FlexSheet control. You can achieve this by setting the TabItemShape property of the C1FlexSheet control.

In Design View

1. Select the **C1FlexSheet** control.
2. Navigate to the **Properties window** and locate the TabItemShape property in Miscellaneous drop-down section.
3. Click the **TabItemShape** drop-down menu and select one of the geometric shapes to apply to the tabs. In our case, we have selected the **Sloped** option.



In XAML

You can customize the shape of tabs in XAML View using the following code:

XAML copyCode

```
<c1:C1FlexSheet x:Name="flexsheet3" BorderBrush="Gray" BorderThickness="1"
Grid.Row="2" Width="1000" TabItemShape="Sloped" HorizontalAlignment="Left"/>
```

In Code

You can also customize the shape of tabs in Code view. To change the tab shape in tab strip, you can add a ComboBox control to your application and use the following code in SelectionChanged event of the ComboBox:

- **Visual Basic**

```
Dim cb As ComboBox = TryCast(sender, ComboBox)
If cb IsNot Nothing Then
    Dim selectedItem = TryCast(cb.SelectedItem, ComboBoxItem)
    If selectedItem IsNot Nothing AndAlso flexsheet3 IsNot Nothing Then
        Select Case selectedItem.Content.ToString()
            Case "Sloped"
                flexsheet3.TabItemShape = C1TabItemShape.Sloped
                Exit Select
            Case "Ribbon"
                flexsheet3.TabItemShape = C1TabItemShape.Ribbon
                Exit Select
            Case "Rounded"
                flexsheet3.TabItemShape = C1TabItemShape.Rounded
                Exit Select
            Case "Rectangle"
                flexsheet3.TabItemShape = C1TabItemShape.Rectangle
                Exit Select
            Case Else
                Exit Select
        End Select
    End If
End If
```

- **C#**

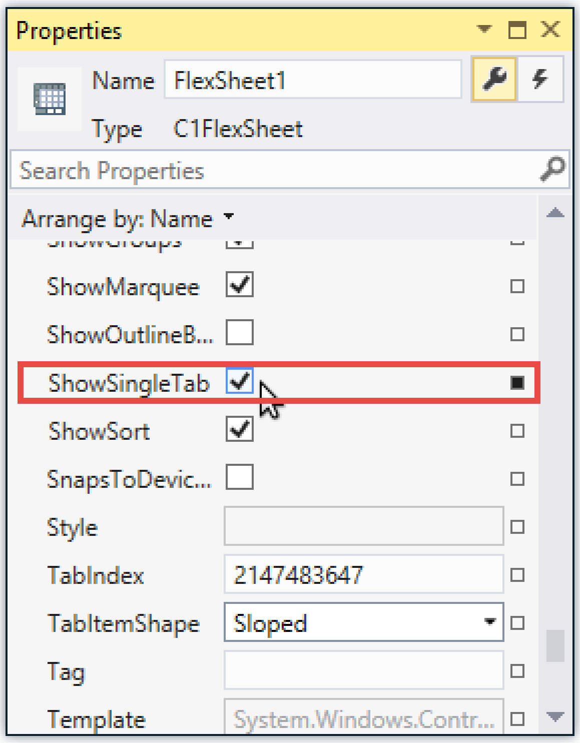
```
ComboBox cb = sender as ComboBox;
if (cb != null)
{
    var selectedItem = cb.SelectedItem as ComboBoxItem;
    if (selectedItem != null && flexsheet3 != null)
    {
        switch (selectedItem.Content.ToString())
        {
            case "Sloped":
                flexsheet3.TabItemShape = C1TabItemShape.Sloped;
                break;
            case "Ribbon":
                flexsheet3.TabItemShape = C1TabItemShape.Ribbon;
                break;
            case "Rounded":
                flexsheet3.TabItemShape = C1TabItemShape.Rounded;
                break;
            case "Rectangle":
                flexsheet3.TabItemShape = C1TabItemShape.Rectangle;
                break;
            default:
                break;
        }
    }
}
```

Hiding Tab Strip

FlexSheet for WPF provides the ability to hide the tab strip containing multiple sheet tabs using ShowSingleTab property of the C1FlexSheet control.

In Design View

1. Select the **C1FlexSheet** control. Ensure that only one sheet is added to the control.
2. Navigate to the **Properties** window and locate the ShowSingleTab property in Miscellaneous drop-down section
3. Uncheck the **ShowSingleTab** property CheckBox as shown in the image below:



Name	FlexSheet1
Type	C1FlexSheet
ShowMarquee	✓
ShowOutlineB...	☐
ShowSingleTab	✓
ShowSort	✓
SnapsToDevic...	☐
Style	
TabIndex	2147483647
TabItemShape	Sloped
Tag	
Template	System.Windows.Contr...

In XAML

You can hide the tab strip appearing in the FlexSheet control by setting the **ShowSingleTab** property to **False**. Following code illustrates the use of **ShowSingleTab** property:

XAML copyCode

```
<c1:C1FlexSheet x:Name="flexsheet1" BorderBrush="Gray" BorderThickness="1"
                Grid.Row="1" Width="1000"  HorizontalAlignment="Left"
ShowSingleTab="False" EditOptions="Rename" />
```

In Code

You can also set the **ShowSingleTab** property in Code view using the following code:

- **Visual Basic**

```
flexsheet1.ShowSingleTab = False
```

- **C#**

```
flexsheet1.ShowSingleTab = false;
```

Working with C1FlexSheet

C1FlexSheet control is a powerful control as it is based on the C1FlexGrid control. It behaves like a spreadsheet rather than just an unbound control as it has formula support and can support various sheet operations as well as cell operations.

Inserting Worksheets in FlexSheet

FlexSheet for WPF is not limited to a single sheet. Multiple worksheets, with an Excel-like tabbed interface, can be added to the C1FlexSheet control by using AddSheet method. The following lines of codes illustrate how to add worksheets to the C1FlexSheet control:

1. Add C1FlexSheet control to the application using the following code in **XAML View**:

   XAML copyCode

   ```
   <c1:C1FlexSheet x:Name="flex" Margin="0,25,0,0"></c1:C1FlexSheet>
   ```

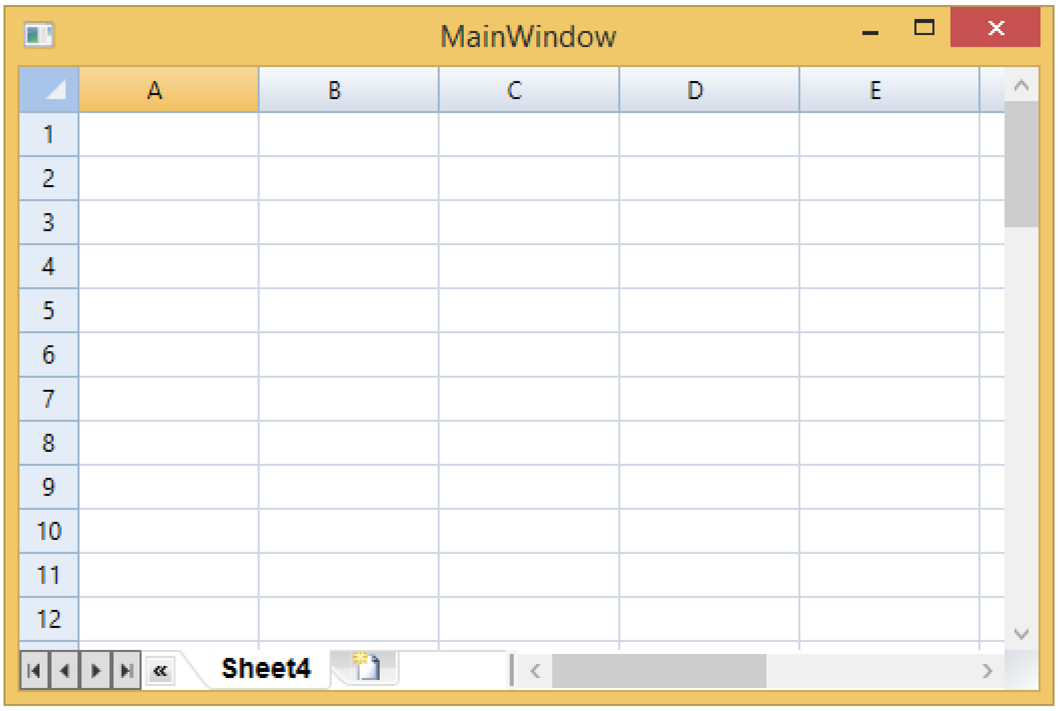
2. Add multiple sheets to the **C1FlexSheet** control by adding the following lines of codes just after the **InitializeComponent()** method in Code view:
   - **Visual Basic**

   ```
   flex.AddSheet("Sheet1", 50, 10)
   ' Sheet 1 with 50 rows and 10 columns
   flex.AddSheet("Sheet2", 20, 10)
   flex.AddSheet("Sheet3", 50, 10)
   ```

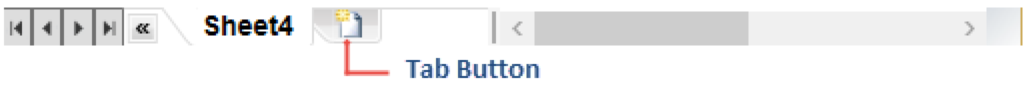
   - **C#**

   ```
   flex.AddSheet("Sheet1", 50, 10);// Sheet 1 with 50 rows and 10 columns
   flex.AddSheet("Sheet2", 20, 10);
   flex.AddSheet("Sheet3", 50, 10);
   flex.AddSheet("Sheet4", 50, 10);
   ```

Here's how a Multi-tabbed FlexSheet looks like:



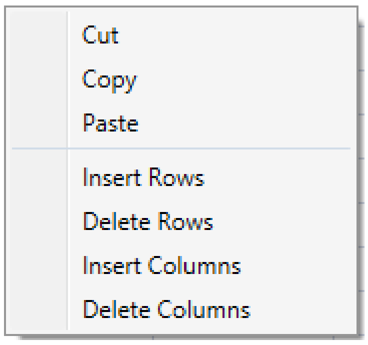
You can also insert sheets or add tabs in C1FlexSheet control at runtime by clicking the **Tab** button on the **Tab Strip**, as shown in the image below:



Sheet Operations

Inserting and Deleting Rows and Columns

Inserting and deleting rows and columns is really easy in **FlexSheet for WPF**. The context menu of C1FlexSheet consists of the options to **Insert Rows, Delete Rows, Insert Columns, Delete Columns** and other clipboard options. These options are members of the ContextMenuCommands enum.



To insert a row, follow the given steps:

1. Right-click on a row or a cell. To insert multiple rows, you need to select multiple rows and then do the right-click operation.
   A context menu appears with the list of options.
2. Select **Insert Rows** option from the context menu.
   The number of row(s) inserted will be equal to the number of selected rows.

To delete a row, follow the given steps:

1. Right-click on a row or a cell. You need to select multiple rows and then do the right-click operation to delete them.
   A context menu appears with the list of options.
2. Select **Delete Rows** option from the context menu.
   The selected row(s) will be deleted.

Similarly, you can insert and delete columns by choosing **Insert Columns** and **Delete Columns** options from the **C1FlexSheet** context menu.

Filtering Columns

When there is a lot of data in a worksheet, it can become a cumbersome task to find information quickly. This is where the **Filters** in C1FlexSheet can be used to abate the data in a worksheet. This makes it easy for you to view just the information you need.

C1FlexSheet provides Excel-style filtering that can be used to filter columns. Filtering in C1FlexSheet is performed using ShowFilterEditor method. The following lines of code in the implementation logic in Code view illustrate filtering:

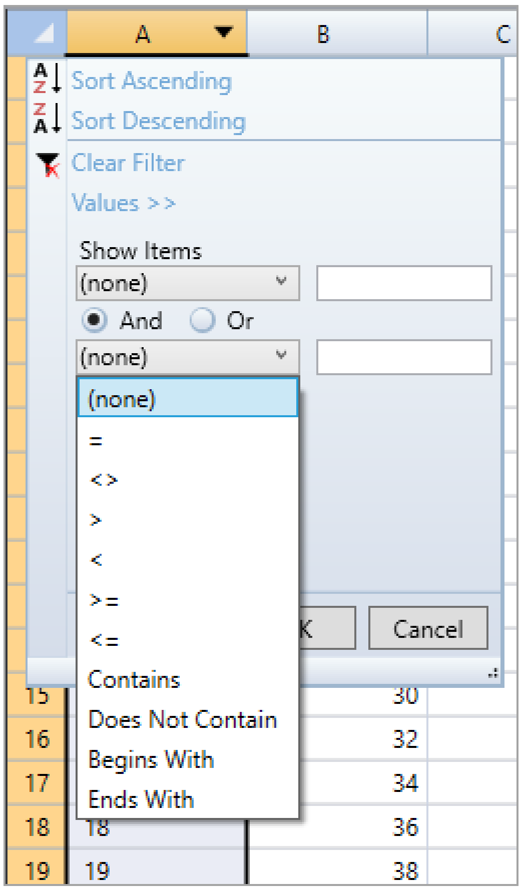
- **Visual Basic**

```
If flex.Columns.Count > 0 Then
    flex.ShowFilterEditor()
```

- **C#**

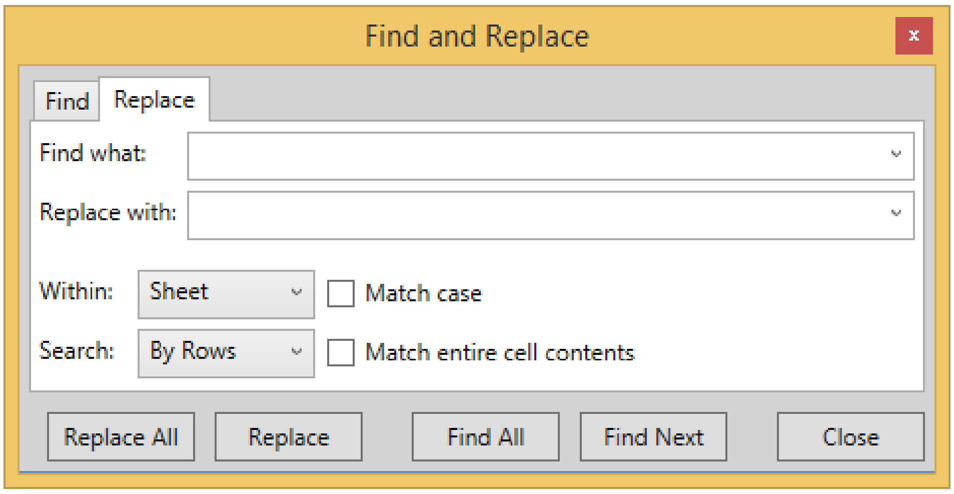
```
if (flex.Columns.Count > 0)
{
    flex.ShowFilterEditor();
}
```

C1FlexSheet allows you to set condition and values in Filters to show specific items on filtering.



Using Find and Replace

FlexSheet for WPF allows you to find and replace any text, numbers or information. You can create a Find and Replace dialog box similar to the dialog box shown in the image below:



FindRange and FindPriority properties of FindOption class can be used to find and replace any text, numbers or information in FlexSheet. The following code creates the objects of FindOption and C1FlexSheet class and illustrates the use of FindRange and FindPriority properties:

- **Visual Basic**

```
Dim _owner As C1FlexSheet
Dim _option As New FindOption(FindRange.Sheet, FindPriority.ByRows, False, False)
```

- **C#**

```
C1FlexSheet _owner;
FindOption _option = new FindOption(FindRange.Sheet, FindPriority.ByRows, false, false);
```

Find text or numbers

Once you create the objects of FindOption and C1FlexSheet class, you can use the following code to find the text or number you want to search in the FlexSheet:

- **Visual Basic**

```
If [String].IsNullOrEmpty(_comboFind.Text) Then
    Return
End If
If Not _comboFind.Items.Contains(_comboFind.Text) Then
    _comboFind.Items.Add(_comboFind.Text)
End If

_owner.FindNext(_comboFind.Text, _option)
```

- **C#**

```
if (String.IsNullOrEmpty(_comboFind.Text))
{
    return;
}
if (!_comboFind.Items.Contains(_comboFind.Text))
    _comboFind.Items.Add(_comboFind.Text);

_owner.FindNext(_comboFind.Text, _option);
```

Replace text or numbers

You can use the following code to replace the text or number in the FlexSheet:

- **Visual Basic**

```
If [String].IsNullOrEmpty(_comboFind.Text) Then
    Return
End If
If Not _comboReplace.Items.Contains(_comboReplace.Text) Then
    _comboReplace.Items.Add(_comboReplace.Text)
End If

Replace(_owner.FindNext(_comboFind.Text, _option))
```

- **C#**

```
if (String.IsNullOrEmpty(_comboFind.Text))
{
    return;
}
if (!_comboReplace.Items.Contains(_comboReplace.Text))
    _comboReplace.Items.Add(_comboReplace.Text);

Replace(_owner.FindNext(_comboFind.Text, _option));
```

Similarly, you can create code to find all the instances of the searched text or number so that when you find all the matching cells, they are displayed in a result list. You can then replace all the searched data in the result list.

Keyboard Navigation

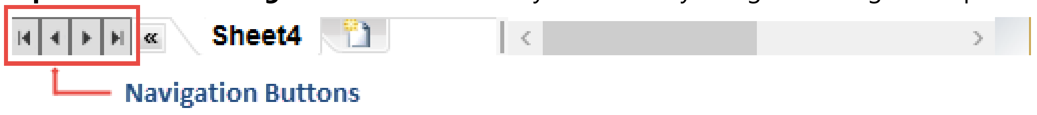
FlexSheet for WPF supports navigation keys that can be used to change the focus on cells and do many other actions in a sheet. You can select cells using the keyboard keys and also make changes to the cell data. The **Keys** on the keyboard and **Actions** performed by them are listed below:

Key	Action
Ctrl+X	Performs cut operation
Ctrl+C	Performs Copy operation
Ctrl+V	Performs Paste operation
Ctrl+Z	Undo
Ctrl+Y	Redo
Delete	Clear
Backspace	Clear and edit
Enter	Navigation down and Edit
Up	Navigation up
Down	Navigation down
Left	Navigation left
Right	Navigation right
PageUp	Navigation to the topmost cell in the column
PageDown	Navigation to the last cell in the column
Home	Navigation to the first cell in the row

End	Navigation to the last cell in the row
Tab	Move to next cell
Shift+ Tab	Move to previous tab
Shift+ Left	Selection Left
Shift+ Right	Selection Right
Shift+ Up	Selection Up
Shift+ Down	Selection Down
Shift+ Home	Selection till the first cell of the row
Shift+ End	Selection till the last cell of the row
Shift+ PageUp	Selection till the first cell of the column
Shift+ PageDown	Selection till the last cell of the column

Tab Navigation

You might struggle while navigating through sheets when there are a lot of worksheets in C1FlexSheet control. It is also difficult to view the names of all the worksheets when there are multiple of them. C1FlexSheet provides you a **Tab Strip** with the **Tab Navigation** buttons so that you can easily navigate through multiple sheets.



These **Navigation buttons** can be of great use in case your **C1FlexSheet** control has multiple sheets in it.

Drag and Drop Rows or Columns

C1FlexSheet allows you to drag and drop rows or columns in a sheet. AllowDragging method is used in C1FlexSheet to drag and drop rows and column. Sometimes, you need to reorder the columns or rows to view their data side-by-side. This reordering can be performed using the **Drag and Drop** in C1FlexSheet.

In XAML

You can implement dragging and dropping of columns in **XAML View** using the following code:

XAML copyCode

```
<c1:C1FlexSheet x:Name="flexsheet" AllowDragging="Columns" AllowSorting="True"
Margin="0,25,0,0"/>
```

Rows can also be dragged and dropped in a worksheet.

When AllowDragging property is used to drag and drop rows in XAML View, sorting is disabled automatically. Following code implements dragging and dropping of rows in **C1FlexSheet** control.

XAML copyCode

```
<c1:C1FlexSheet x:Name="flexsheet" AllowDragging="Rows" />
```

In Code

Following lines of code implement the dragging and dropping of columns in a worksheet:

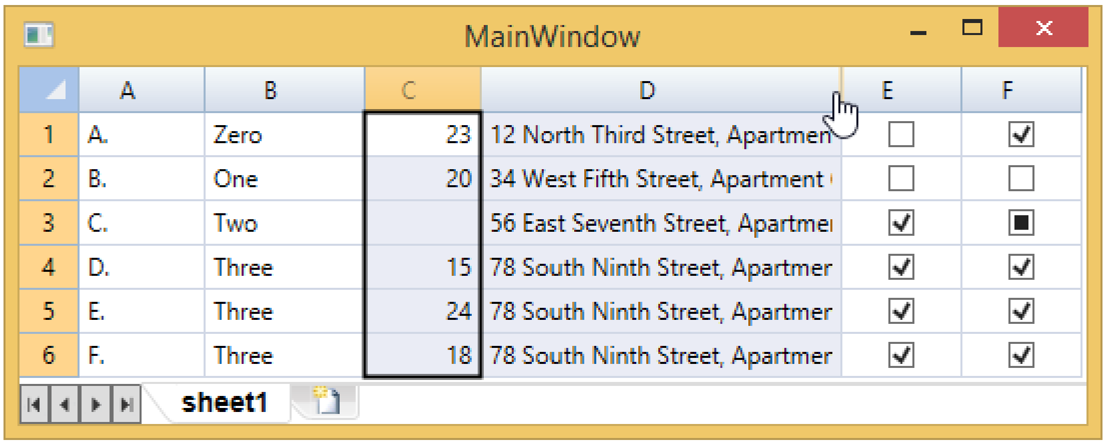
- **Visual Basic**

```
flexsheet.AllowDragging = AllowDragging.Columns
```

- **C#**

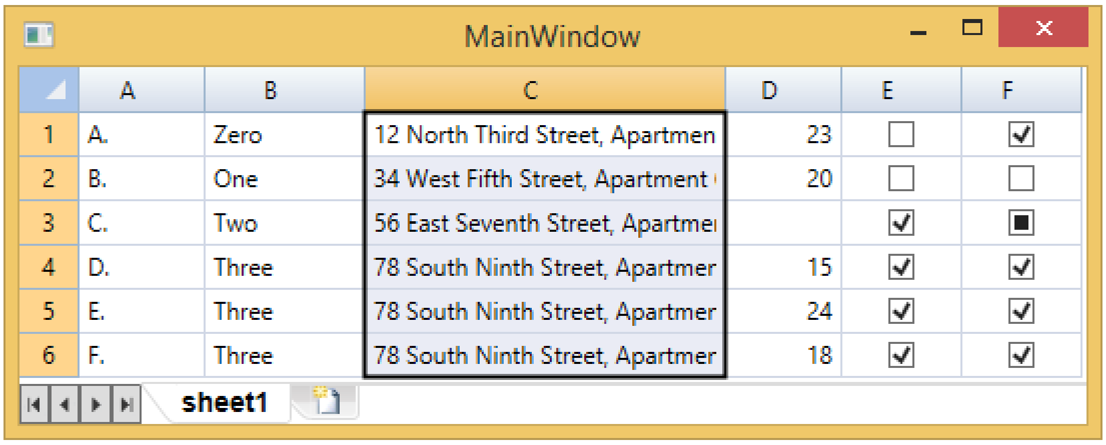
```
flexsheet.AllowDragging = AllowDragging.Columns;
```

For example, you have a list of customer and their details in a sheet and want to reorder the **Age** and **Address** columns. You can drag the **Age** column and drop it after the **Address** column.



	A	B	C	D	E	F
1	A.	Zero	23	12 North Third Street, Apartmen	☐	✓
2	B.	One	20	34 West Fifth Street, Apartment	☐	☐
3	C.	Two		56 East Seventh Street, Apartme	✓	■
4	D.	Three	15	78 South Ninth Street, Apartmer	✓	✓
5	E.	Three	24	78 South Ninth Street, Apartmer	✓	✓
6	F.	Three	18	78 South Ninth Street, Apartmer	✓	✓

After you drop the **Age** Column after the **Address** column, it looks similar to the following:



	A	B	C	D	E	F
1	A.	Zero	12 North Third Street, Apartmen	23	☐	✓
2	B.	One	34 West Fifth Street, Apartment	20	☐	☐
3	C.	Two	56 East Seventh Street, Apartme		✓	■
4	D.	Three	78 South Ninth Street, Apartmer	15	✓	✓
5	E.	Three	78 South Ninth Street, Apartmer	24	✓	✓
6	F.	Three	78 South Ninth Street, Apartmer	18	✓	✓

AllowDragging property can also be applied to the rows in Code view. The following code shows how to implement the drag and drop functionality on rows. However, sorting needs to be disabled to drag and drop rows.

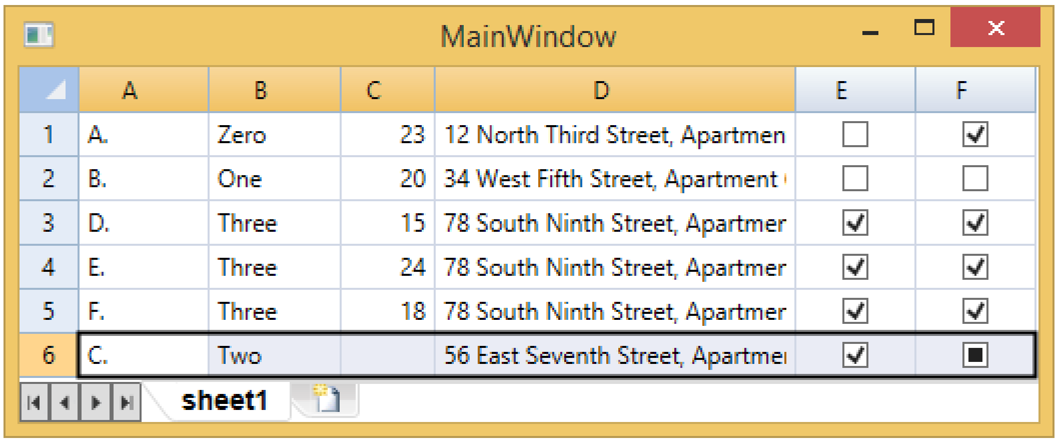
- **Visual Basic**

```
flexsheet.AllowDragging = AllowDragging.Rows
```

- **C#**

```
flexsheet.AllowDragging = AllowDragging.Rows;
flexsheet.AllowSorting = false;
```

When you drag and drop a row, it will look similar to the image given below. In our case, we dragged the third row and dropped it after the sixth row.



	A	B	C	D	E	F
1	A.	Zero	23	12 North Third Street, Apartmen	☐	✓
2	B.	One	20	34 West Fifth Street, Apartment	☐	☐
3	D.	Three	15	78 South Ninth Street, Apartmer	✓	✓
4	E.	Three	24	78 South Ninth Street, Apartmer	✓	✓
5	F.	Three	18	78 South Ninth Street, Apartmer	✓	✓
6	C.	Two		56 East Seventh Street, Apartmei	✓	■

Data Binding

You can easily perform data binding in **FlexSheet**. Here, we will discuss about two types of data binding: Data binding using data source and data binding using IEnumerable Interface.

Data Binding using Data Source

To bind C1FlexSheet to a data source, we have used **C1NWind.mdb** database. You can find the **C1NWind.mdb** database in the **Documents\ComponentOne Samples\Common** directory.

The following code binds the C1FlexSheet control to the **C1NWind.mdb** database:

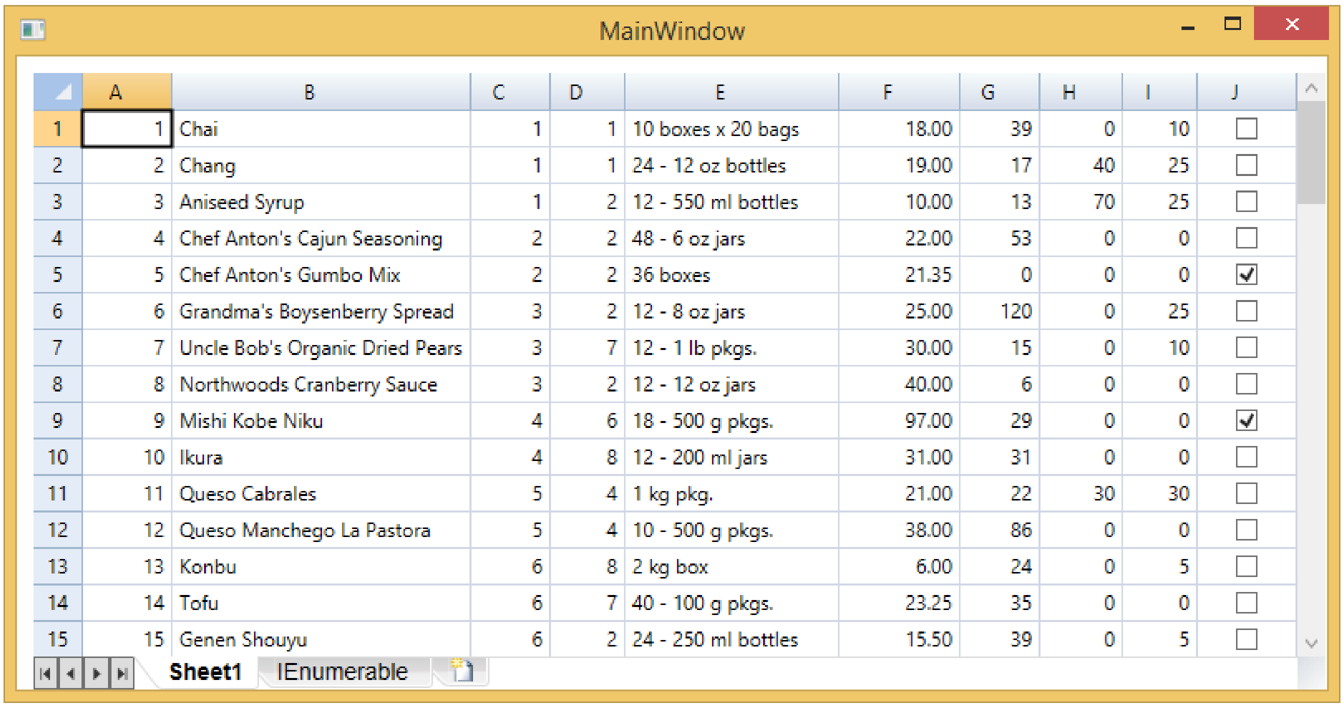
- **Visual Basic**

```
Dim con As New System.Data.OleDb.OleDbConnection("Provider=Microsoft.Jet.OLEDB.4.0;Data
        Source=C:\Users\Windows 8.1\Documents\ComponentOne Samples\Common\C1NWind.mdb")
Dim adpt As New System.Data.OleDb.OleDbDataAdapter("SELECT * FROM Products", con)
Dim dt As New System.Data.DataTable()
adpt.Fill(dt)
_flexSheet.AddSheet("Sheet1", dt.DefaultView)
```

- **C#**

```
OleDbConnection con = new OleDbConnection(@"Provider=Microsoft.Jet.OLEDB.4.0;Data
   Source=C:\Users\Windows 8.1\Documents\ComponentOne Samples\Common\C1NWind.mdb");
OleDbDataAdapter adpt = new OleDbDataAdapter("SELECT * FROM Products", con);
DataTable dt = new DataTable();
adpt.Fill(dt);
_flexSheet.AddSheet("Sheet1", dt.DefaultView);
```

The above code retrieves the data from the **Products** table of **C1NWind.mdb** database. The data reflected in the FlexSheet after binding with data source is shown in the image below:


MainWindow

	A	B	C	D	E	F	G	H	I	J
1	1	Chai	1	1	10 boxes x 20 bags	18.00	39	0	10	☐
2	2	Chang	1	1	24 - 12 oz bottles	19.00	17	40	25	☐
3	3	Aniseed Syrup	1	2	12 - 550 ml bottles	10.00	13	70	25	☐
4	4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	22.00	53	0	0	☐
5	5	Chef Anton's Gumbo Mix	2	2	36 boxes	21.35	0	0	0	☑
6	6	Grandma's Boysenberry Spread	3	2	12 - 8 oz jars	25.00	120	0	25	☐
7	7	Uncle Bob's Organic Dried Pears	3	7	12 - 1 lb pkgs.	30.00	15	0	10	☐
8	8	Northwoods Cranberry Sauce	3	2	12 - 12 oz jars	40.00	6	0	0	☐
9	9	Mishi Kobe Niku	4	6	18 - 500 g pkgs.	97.00	29	0	0	☑
10	10	Ikura	4	8	12 - 200 ml jars	31.00	31	0	0	☐
11	11	Queso Cabrales	5	4	1 kg pkg.	21.00	22	30	30	☐
12	12	Queso Manchego La Pastora	5	4	10 - 500 g pkgs.	38.00	86	0	0	☐
13	13	Konbu	6	8	2 kg box	6.00	24	0	5	☐
14	14	Tofu	6	7	40 - 100 g pkgs.	23.25	35	0	0	☐
15	15	Genen Shouyu	6	2	24 - 250 ml bottles	15.50	39	0	5	☐

Sheet1 | IEnumerable

Data Binding using IEnumerable<T> Interface

IEnumerable<T> interface inherits from the **IEnumerable** interface. To bind the data using **IEnumerable <T>** interface, a user-defined class needs to be added in the code. This class must implement the **IEnumerable** interface. In our case, we have created a class named **Customer** to implement the **IEnumerable** interface.

The following code implements the IEnumerable interface:

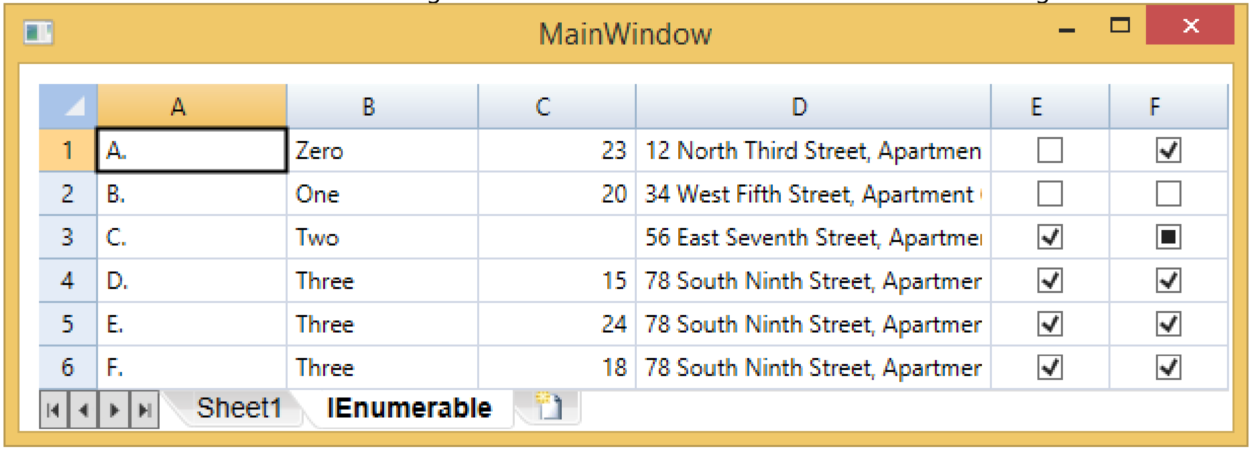
- **Visual Basic**

```
Dim result As IEnumerable(Of Customer) = Customer.GetSampleCustomerList()
Dim cus = result.Where(Function(x) x.LastName = "Three")
_flexSheet.AddSheet("IEnumerable", result)
```

- **C#**

```
IEnumerable<Customer> result = Customer.GetSampleCustomerList();
var cus = result.Where(x => x.LastName == "Three");
_flexSheet.AddSheet("IEnumerable", result);
```

The above code retrieves the data from a list of customers and their details defined in Customers class. The data reflected in the FlexSheet after binding with **IEnumerable<T>** interface is shown in the image below:


MainWindow

	A	B	C	D	E	F
1	A.	Zero	23	12 North Third Street, Apartmen	☐	☑
2	B.	One	20	34 West Fifth Street, Apartment	☐	☐
3	C.	Two		56 East Seventh Street, Apartme	☑	■
4	D.	Three	15	78 South Ninth Street, Apartmer	☑	☑
5	E.	Three	24	78 South Ninth Street, Apartmer	☑	☑
6	F.	Three	18	78 South Ninth Street, Apartmer	☑	☑

Sheet1 | IEnumerable

Data Validation

There are a lot of ways to validate data in FlexSheet. We will explain a simple way of validation, which is throwing an exception on passing an invalid value.

To validate data, we have created a class named ProductBase which contains data about product line, color, name, price, cost, weight, volume discontinued, and rating. Another class named ProductRow contains the **SetValue** method which simply throws exceptions when a property setter is passed an invalid value. This method allows for property-level validation only (no item-level validation). The following code implements data validation in SetValue method:

- **Visual Basic**

```
    Try
        If p = "Price" AndAlso DirectCast(value, System.Nullable(Of Double)) <= 0 Then
            Throw New Exception("Price must be > 0.")
        End If
        If p = "Cost" AndAlso DirectCast(value, System.Nullable(Of Double)) <= 0 Then
            Throw New Exception("Cost must be > 0.")
        End If
        If p = "Weight" AndAlso DirectCast(value, System.Nullable(Of Double)) <= 0 Then
            Throw New Exception("Weight must be > 0.")
        End If
        If p = "Rating" AndAlso (DirectCast(value, System.Nullable(Of Integer)) < 0 _
               OrElse DirectCast(value, System.Nullable(Of Integer)) > 5) Then
            Throw New Exception("Rating must be between 0 and 5.")
        End If
        MyBase.SetValue(p, value)
    Catch e As Exception
        MessageBox.Show(e.Message)
    End Try
End Sub
```

- **C#**

```
try
{
    if (p == "Price" && (double?)value <= 0)
    {
        throw new Exception("Price must be > 0.");
    }
    if (p == "Cost" && (double?)value <= 0)
    {
        throw new Exception("Cost must be > 0.");
    }
    if (p == "Weight" && (double?)value <= 0)
    {
        throw new Exception("Weight must be > 0.");
    }
    if (p == "Rating" && ((int?)value < 0 || (int?)value > 5))
    {
        throw new Exception("Rating must be between 0 and 5.");
    }
    base.SetValue(p, value);
}
catch(Exception e)
{
    MessageBox.Show(e.Message);
}
```

The ProductThrow class throws exception when a user sets price, cost or weight properties to a negative value or sets rating to a value less than zero and greater than 5.

Freezing and Unfreezing Rows and Columns

You might want to compare data in a worksheet while working on it. However, if the data in the worksheet is in large amount then it becomes difficult to compare. C1FlexSheet allows you to freeze rows and columns that can help you compare different parts of your data easily.

When you want to compare data in specific rows/columns and want that data fixed at a place in a worksheet while scrolling through the rest of the data in it, you can freeze the rows or columns using Frozen property in C1FlexSheet. Also, when you are done comparing the data, you can unfreeze the rows and columns using Frozen property. You can use the following lines of code to freeze and unfreeze the data in a sheet:

- **Visual Basic**

```
If flex.Rows.Frozen > 0 OrElse flex.Columns.Frozen > 0 Then
    ' unfreeze
    For i As Int32 = 0 To flex.Rows.Frozen - 1
        flex.Rows(i).Visible = True
    Next
    For i As Int32 = 0 To flex.Columns.Frozen - 1
        flex.Columns(i).Visible = True
    Next
    flex.Rows.Frozen = 0
    flex.Columns.Frozen = 0
Else
    ' freeze
    Dim vr = flex.ViewRange
    For i As Int32 = 0 To vr.TopRow - 1
        flex.Rows(i).Visible = False
    Next
    For i As Int32 = 0 To vr.LeftColumn - 1
        flex.Columns(i).Visible = False
    Next
    flex.Rows.Frozen = flex.Selection.TopRow
    flex.Columns.Frozen = flex.Selection.LeftColumn
    flex.ScrollIntoView(flex.Rows.Frozen, flex.Columns.Frozen)
End If
```

- **C#**

```
if (flex.Rows.Frozen > 0 || flex.Columns.Frozen > 0)
{
    // unfreeze
    for (var i = 0; i < flex.Rows.Frozen; i++)
    {
        flex.Rows[i].Visible = true;
    }
    for (var i = 0; i < flex.Columns.Frozen; i++)
    {
        flex.Columns[i].Visible = true;
    }
    flex.Rows.Frozen = 0;
    flex.Columns.Frozen = 0;
}
else
{
    // freeze
    var vr = flex.ViewRange;
    for (var i = 0; i < vr.TopRow; i++)
    {
        flex.Rows[i].Visible = false;
    }
    for (var i = 0; i < vr.LeftColumn; i++)
    {
        flex.Columns[i].Visible = false;
    }
    flex.Rows.Frozen = flex.Selection.TopRow;
    flex.Columns.Frozen = flex.Selection.LeftColumn;
    flex.ScrollIntoView(flex.Rows.Frozen, flex.Columns.Frozen);
}
```

Grouping and Ungrouping Rows

FlexSheet for WPF provides you the ease to group data in a sheet where you might have a huge amount of data and want to organize it. Data can easily be organized in groups using GroupRows method in C1FlexSheet control that allows you to divide the data and show or hide different sections of the worksheet.

Following lines of code illustrate the use of GroupRows method for grouping rows in C1FlexSheet control:

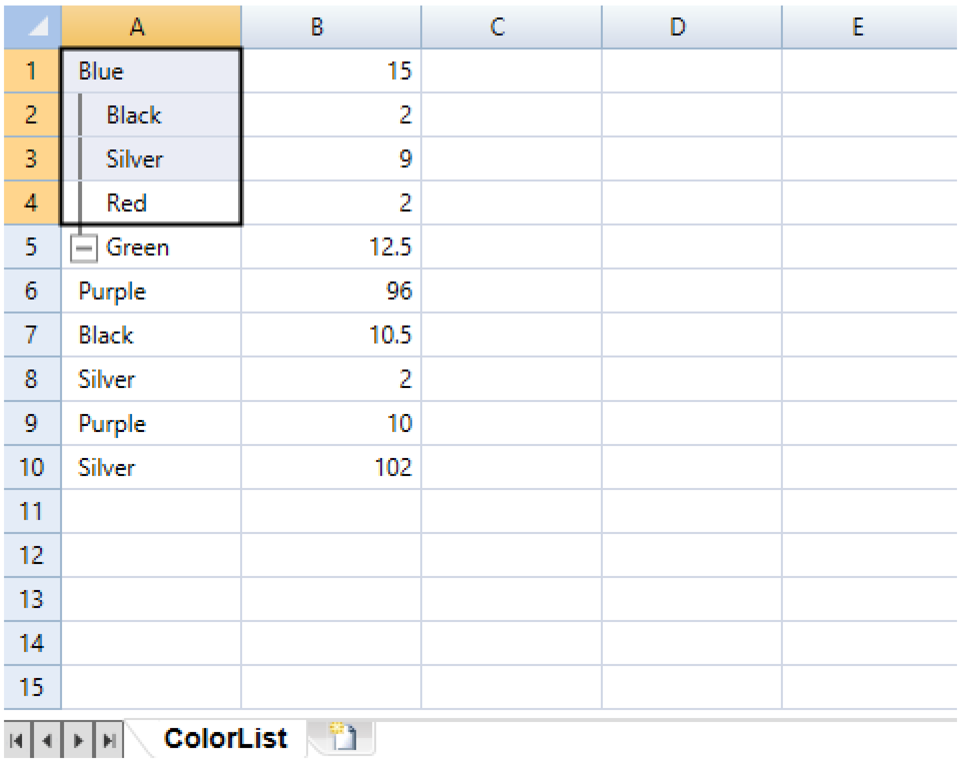
- **Visual Basic**

```
flex.GroupRows(flex.Selection)
```

- **C#**

```
flex.GroupRows(flex.Selection);
```

On applying grouping, the output will look similar to the image given below. In this example we have grouped first four rows in a worksheet.



	A	B	C	D	E
1	Blue	15			
2	Black	2			
3	Silver	9			
4	Red	2			
5	Green	12.5			
6	Purple	96			
7	Black	10.5			
8	Silver	2			
9	Purple	10			
10	Silver	102			
11					
12					
13					
14					
15					

If you want to Ungroup rows, you can use the UnGroupRows method illustrated in the following code:

- **C#**

```
flex.UnGroupRows();
```

Removing Duplicate Rows

When you work with large amount of data in a sheet, you might encounter duplicate rows or might accidentally copy few rows unknowingly. In this situation, finding and deleting such duplicate rows can be time-consuming. To get out of this situation, you can use the RemoveDuplicates method in C1FlexSheet control.

RemoveDuplicates method is used in C1FlexSheet to remove rows that consist of same data. The following lines of code illustrate removing duplicate rows in the selected columns:

- **Visual Basic**

```
Dim selectedColumns As New List(Of Integer)()
For i As Integer = flex.Selection.LeftColumn To flex.Selection.RightColumn
    selectedColumns.Add(i)
Next
flex.RemoveDuplicates(flex.Selection, selectedColumns)
```

- **C#**

```
List<int> selectedColumns = new List<int>();
for (int i = flex.Selection.LeftColumn; i <= flex.Selection.RightColumn; i++)
{
    selectedColumns.Add(i);
}
flex.RemoveDuplicates(flex.Selection, selectedColumns);
```

Protect Sheets and Cell Locking

You can protect sheets and lock cells in **FlexSheet for WPF**. IsProtected property is used to protect sheets in C1FlexSheet control. The following code illustrates the use of IsProtected property:

- **Visual Basic**

```
protectedSheet.IsProtected = True
```

- **C#**

```
protectedSheet.IsProtected = true;
```

AddLockedCell method can be used to lock cells in C1FlexSheet control. The AddLockedCell method locks the specified cell range, as illustrated in the following code:

- **Visual Basic**

```
protectedSheet.AddLockedCell(0, 0, 1, 1)
```

- **C#**

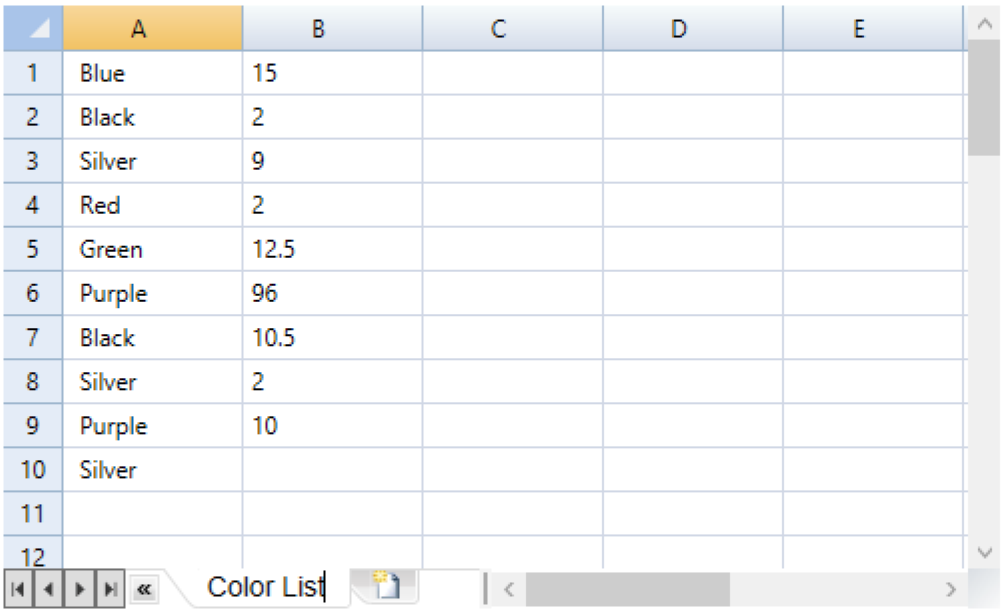
```
protectedSheet.AddLockedCell(0, 0, 1, 1);
```

Sheet Renaming

Renaming a sheet in **FlexSheet for WPF** is as easy as it can be. To rename a sheet in C1FlexSheet, you just need to follow these simple steps:

1. Right-click the Tab you need to rename, on the tab strip.
   A context menu will appear.

2. Select **Rename** from the context menu.
3. Rename the sheet with the name of your choice. In the following example, we renamed the sheet to **Color List**.

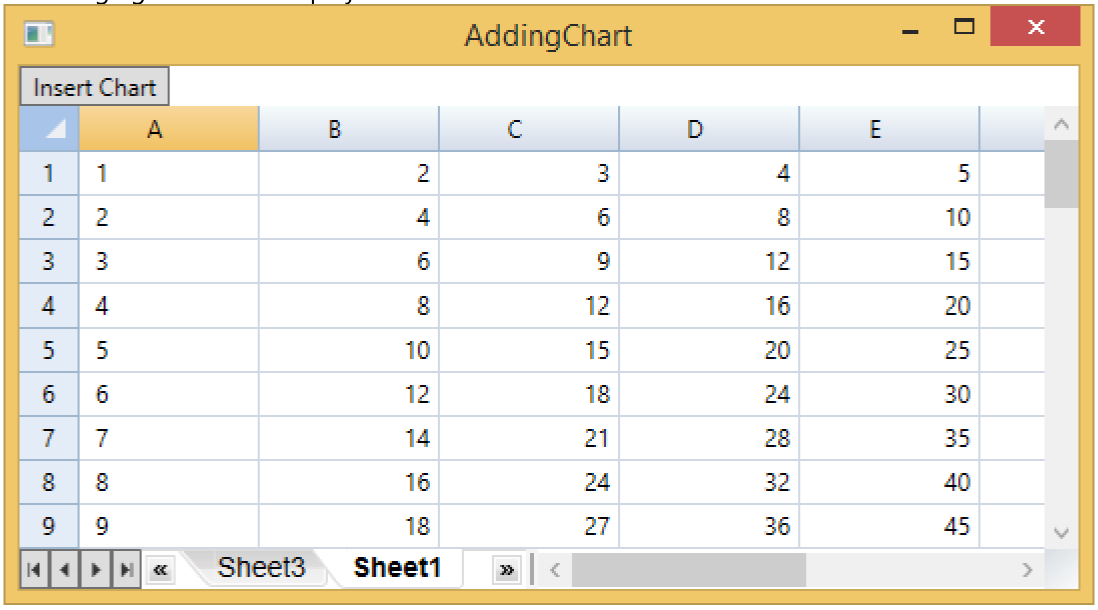


	A	B	C	D	E
1	Blue	15			
2	Black	2			
3	Silver	9			
4	Red	2			
5	Green	12.5			
6	Purple	96			
7	Black	10.5			
8	Silver	2			
9	Purple	10			
10	Silver				
11					
12					

Sheet Reordering

You can also reorder sheets in C1FlexSheet control. If you have multiple sheets in C1FlexSheet control and want to change the order of sheets then simply drag the required sheet's tab to the position where you want to place it.

The image given below displays the reordered sheets where **Sheet1** is shifted after **Sheet3** at runtime:



	A	B	C	D	E
1	1	2	3	4	5
2	2	4	6	8	10
3	3	6	9	12	15
4	4	8	12	16	20
5	5	10	15	20	25
6	6	12	18	24	30
7	7	14	21	28	35
8	8	16	24	32	40
9	9	18	27	36	45

Sorting Columns

Sorting is an important requirement when it comes to listing data alphabetically and arranging data in ascending or descending order. C1FlexSheet uses AllowSorting property to sort data in a worksheet.

In XAML

You can easily sort data using AllowSorting property in XAML View. The following code illustrates the use of AllowSorting property:

XAML copyCode

```
<c1:C1FlexSheet x:Name="flexsheet" AllowDragging="Columns" AllowSorting="True"
Margin="0,25,0,0"/>
```

In Code

The following code illustrates the use of AllowSorting property in Code view:

- **Visual Basic**

```
flexsheet.AllowSorting = True
```

- **C#**

```
flexsheet.AllowSorting = true;
```

The C1FlexSheet control also provides range-based unbound sorting. Sorting in FlexSheet can be done by using SortDialog class. The following lines of code show the unbound sorting in **C1FlexSheet** control:

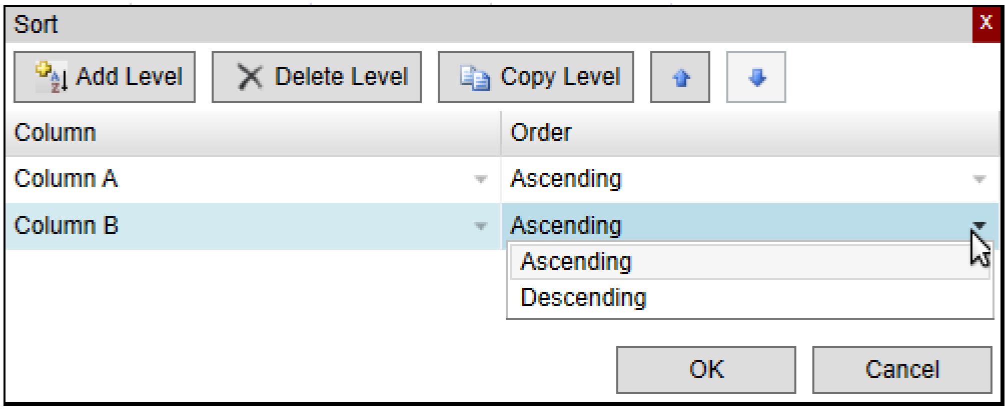
- **Visual Basic**

```
If flexsheet.Columns.Count > 0 Then
    Dim sortDialog = New SortDialog(flexsheet)
    sortDialog.Show()
End If
```

- **C#**

```
if (flexsheet.Columns.Count > 0)
{
    var sortDialog = new SortDialog(flexsheet);
    sortDialog.Show();
}
```

The Sort dialog box provides options of adding, deleting, and copying levels. You can specify columns and their respective orders (ascending or descending) for sorting data in these levels.



Column	Order
Column A	Ascending
Column B	Ascending

Undo Redo Operations

C1FlexSheet allows you to use Undo method to undo the last action and Redo method to repeat the last action performed.

Following code illustrates the use of Undo method:

- **Visual Basic**

```
flex.Undo()
```

- **C#**

```
flex.Undo();
```

Following code illustrates the use of Redo method:

- **Visual Basic**

```
flex.Redo()
```

- **C#**

```
flex.Redo();
```

Cell Operations

Adding Comments to a Cell

When you have some additional information related to the data in a cell and cannot include that information in the cell, then you can insert a comment in it. C1FlexSheet control uses InsertComment method to insert comment in a cell without any distortion in the already existing data.

Following lines of code are used to implement InsertComment method in C1FlexSheet:

- **Visual Basic**

```
If flex.Selection.IsValid AndAlso flex.Selection.IsSingleCell Then
    flex.InsertComment(flex.Selection)
End If
```

- **C#**

```
if (flex.Selection.IsValid && flex.Selection.IsSingleCell)
{
    flex.InsertComment(flex.Selection);
}
```

Clipboard Operations

All the Clipboard operations, Cut,Copy, and Paste commands are supported by C1FlexSheet controls. You can easily cut, copy, or paste data in FlexSheet.

Cut Operation

The following code performs the cut operation in C1FlexSheet control:

- **Visual Basic**

```
' create undoable cell range action and record the cell values
' of the current selection before changing them
Dim action = New CellRangeEditAction(flex)
```

```
flex.Copy()
For Each cell In flex.Selection.Cells
    Try
        flex(cell.Row, cell.Column) = Nothing
    Catch
    End Try
Next

' record the cell values after the changes and add the
' undoable action to the undo stack
If action.SaveNewState() Then
    flex.UndoStack.AddAction(action)
End If
```

- **C#**

```
// create undoable cell range action and record the cell values
// of the current selection before changing them
var action = new CellRangeEditAction(flex);

flex.Copy();
foreach (var cell in flex.Selection.Cells)
{
    try
    {
        flex[cell.Row, cell.Column] = null;
    }
    catch { }
}

// record the cell values after the changes and add the
// undoable action to the undo stack
if (action.SaveNewState())
{
    flex.UndoStack.AddAction(action);
}
```

The above code refers a class named **CellRangeEditAction** which includes the implementation of recording the values of all the cells within the current selection of the control.

Copy operation

Data from cell(s) can easily be copied in C1FlexSheet control using Copy method. The following code uses Copy method to copy the data from the selected cells:

- **Visual Basic**

```
flex.Copy()
```

- **C#**

```
flex.Copy();
```

Paste operation

Data from cell(s) can easily be pasted in C1FlexSheet control using Paste method. The following code uses Paste method to paste the copied data:

- **Visual Basic**

```
flex.Paste()
```

- **C#**

```
flex.Paste();
```

Importing and Exporting

FlexSheet for WPF enables you to import and export files with different formats. You can import data from Excel 97-2003 Workbook (.xls), Excel Workbook (.xlsx), and Text File (.txt) formats and save it in C1FlexSheet control. Data from C1FlexSheet can be exported to Excel 97-2003 Workbook (.xls), Excel Workbook (.xlsx), Text File (.txt), HTML File (.htm or.html), Comma Separated Values (.csv), and PDF File(.pdf) formats.

Importing FlexSheet

C1FlexSheet allows import of Excel files (.xls, .xlsx) and text file (.txt). This functionality is attained by using ImportFileFormat enum. The following code sample illustrates importing Excel files in C1FlexSheet:

- **Visual Basic**

```
Dim dlg = New Microsoft.Win32.OpenFileDialog()
dlg.Filter = "Excel 97-2003 Workbook (*.xls)|*.xls|" +
             "Excel Workbook (*.xlsx)|*.xlsx" +
             "Text File (*.txt)|*.txt|"

If dlg.ShowDialog().Value Then

    Try
        Using s = dlg.OpenFile()

            Dim ext = System.IO.Path.GetExtension(dlg.SafeFileName).ToLower()
            Select Case ext
                Case ".txt"
                    flex.Load(s, ImportFileFormat.TXT)
                    Exit Select
                Case ".xlsx"
                    flex.Load(s, ImportFileFormat.XLSX)
                    ' ImportFileFormat uses namespace FlexGrid
                    Exit Select
                Case ".xls"
                    flex.Load(s, ImportFileFormat.XLS)
                    Exit Select

            End Select

        End Using

    Catch x As Exception
        Dim msg = "Error opening file: " & vbCr & vbLf & vbCr & vbLf + x.Message
        MessageBox.Show(msg, "Error", MessageBoxButton.OK)

    End Try
End If
```

- **C#**

```
var dlg = new Microsoft.Win32.OpenFileDialog();
dlg.Filter = "Excel 97-2003 Workbook (*.xls)|*.xls|"
```

```
    + "Excel Workbook (*.xlsx)|*.xlsx" +
    "Text File (*.txt)|*.txt|";

if (dlg.ShowDialog().Value)
{

    try
    {
        using (var s = dlg.OpenFile())
        {

            var ext = System.IO.Path.GetExtension(dlg.SafeFileName).ToLower();
            switch (ext)
            {
                case ".txt":
                    flex.Load(s, ImportFileFormat.TXT);
                    break;
                case ".xlsx":
                    // ImportFileFormat uses namespace FlexGrid
                    flex.Load(s, ImportFileFormat.XLSX);
                    break;
                case ".xls":
                    flex.Load(s, ImportFileFormat.XLS);
                    break;
            }

        }

    }

    catch (Exception x)
    {
        var msg = "Error opening file: \r\n\r\n" + x.Message;
        MessageBox.Show(msg, "Error", MessageBoxButton.OK);
    }

}
```

Exporting FlexSheet

FlexSheet can be exported to .xls, .xlsx, .htm, .html, .csv, and .txt file formats using FileFormat and SaveOptions enums. It also exports .pdf file format using PdfExportOptions class, which provides options for pdf export and **SavePDF** method, which saves the grid to a PDF stream. The following steps illustrate exporting C1FlexSheet to these formats:

- **Visual Basic**

```
Dim dlg = New Microsoft.Win32.SaveFileDialog()
dlg.DefaultExt = "xlsx"
dlg.Filter = "Excel Workbook (*.xlsx)|*.xlsx|" +
             "Excel 97-2003 Workbook (*.xls)|*.xls|" +
             "HTML File (*.htm;*.html)|*.htm;*.html|" +
             "Comma Separated Values (*.csv)|*.csv|" +
             "Text File (*.txt)|*.txt|" + "PDF (*.pdf)|*.pdf"

If dlg.ShowDialog().Value Then
    Using s = dlg.OpenFile()
        Dim ext = System.IO.Path.GetExtension(dlg.SafeFileName).ToLower()
        Select Case ext
            Case ".htm", ".html"
                flex.Save(s, FileFormat.Html, SaveOptions.Formatted)
                Exit Select
            Case ".csv"
                flex.Save(s, FileFormat.Csv, SaveOptions.Formatted)
                Exit Select
            Case ".txt"
                flex.Save(s, FileFormat.Text, SaveOptions.Formatted)
                Exit Select
            Case ".pdf"
                SavePdf(s, "ComponentOne ExcelBook")
                Exit Select
            Case ".xlsx"
                flex.SaveXlsx(s)
                Exit Select
            Case Else
                flex.SaveXls(s)
                Exit Select
        End Select
    End Using
End If
```

- **C#**

```
var dlg = new Microsoft.Win32.SaveFileDialog();
dlg.DefaultExt = "xlsx";
dlg.Filter =
    "Excel Workbook (*.xlsx)|*.xlsx|" +
    "Excel 97-2003 Workbook (*.xls)|*.xls|" +
    "HTML File (*.htm;*.html)|*.htm;*.html|" +
    "Comma Separated Values (*.csv)|*.csv|" +
    "Text File (*.txt)|*.txt|" +
    "PDF (*.pdf)|*.pdf";

if (dlg.ShowDialog().Value)
{
    using (var s = dlg.OpenFile())
    {
        var ext = System.IO.Path.GetExtension(dlg.SafeFileName).ToLower();
        switch (ext)
        {
            case ".htm":
            case ".html":
                flex.Save(s, FileFormat.Html, SaveOptions.Formatted);
                break;
            case ".csv":
                flex.Save(s, FileFormat.Csv, SaveOptions.Formatted);
                break;
            case ".txt":
                flex.Save(s, FileFormat.Text, SaveOptions.Formatted);
                break;
            case ".pdf":
                SavePdf(s, "ComponentOne ExcelBook");
                break;
            case ".xlsx":
                flex.SaveXlsx(s);
                break;
            default:
                flex.SaveXls(s);
                break;
        }
    }
}
```

The implementation of the **SavePDF** method used in the above code is given in the following code:

- **Visual Basic**

```
Private Sub SavePdf(s As Stream, documentName As String)
    Dim options As New PdfExportOptions()
    options.Margin = New Thickness(96, 96, 96 / 2, 96 / 2)
    options.ScaleMode = ScaleMode.ActualSize
    flex.SavePdf(s, options)
    s.Close()
```

- **C#**

```
void SavePdf(Stream s, string documentName)
{
    PdfExportOptions options = new PdfExportOptions();
    options.Margin = new Thickness(96, 96, 96 / 2, 96 / 2);
    options.ScaleMode = ScaleMode.ActualSize;
    flex.SavePdf(s, options);
    s.Close();
}
```

Using Formulas

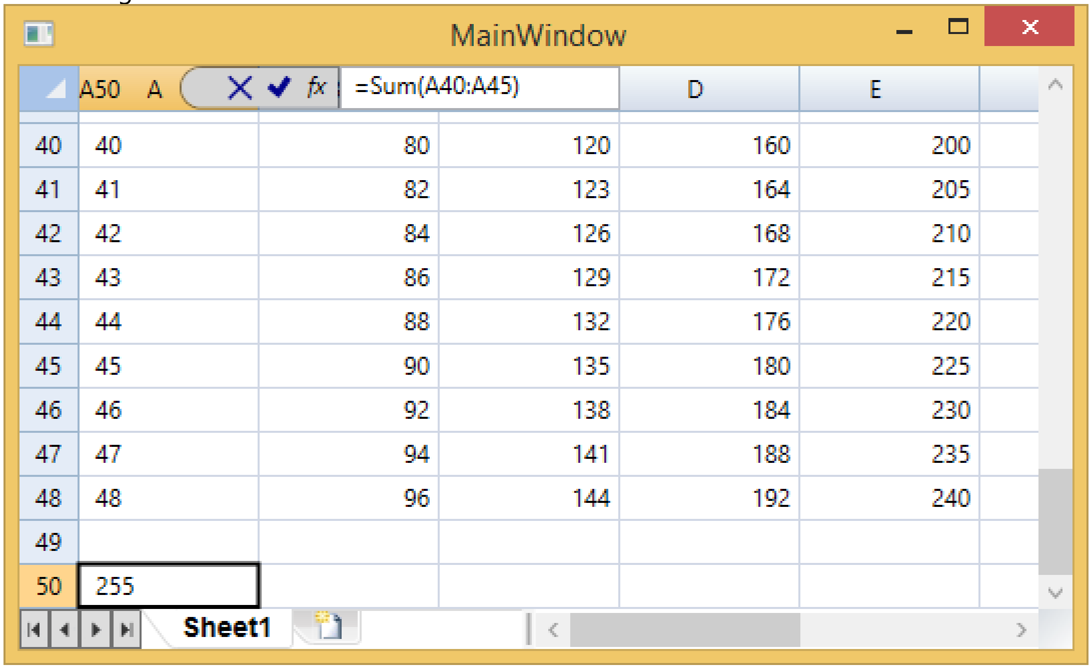
To perform calculations in cells, you can enter formulas in the FormulaBar control bound with the FlexSheet control. Binding the C1FormulaBar with C1FlexSheet enables you to perform calculations using Excel-style formulas.

You can bind C1FormulaBar with C1FlexSheet using the following code:

XAML copyCode

```
<c1:C1FormulaBar Name="formula_Bar" Width="300" HorizontalAlignment="Left"
VerticalAlignment="Top" FlexSheet="{Binding ElementName=flex, Mode=OneWay}"/>
```

Enter a simple function such as SUM at run time to add values of different cells and see the use of formulas as shown in the image below:



=Sum(A40:A45)

	A			D	E
40	40	80	120	160	200
41	41	82	123	164	205
42	42	84	126	168	210
43	43	86	129	172	215
44	44	88	132	176	220
45	45	90	135	180	225
46	46	92	138	184	230
47	47	94	141	188	235
48	48	96	144	192	240
49					
50	255				

Printing Data

FlexSheet for WPF allows you to print the data. As it is always better to view the print preview before printing the data, C1FlexSheet shows the print preview of the data before you print it. Print preview allows you to see how the data will look on printing.

Print Preview

C1FlexSheet uses PrintPreview method to print the data in a worksheet. The parameters of this method allows you to select the document name, scale mode, margin, and maximum number of pages. The code below uses the PrintPreview method to provide the preview before printing so that you can view the layout and then print the data:

- **Visual Basic**

```
Dim scaleMode__1 = ScaleMode.PageWidth
flex.PrintPreview("C1FlexSheet", scaleMode__1, New Thickness(96), Integer.MaxValue)
```

- **C#**

```
var scaleMode = ScaleMode.PageWidth;
flex.PrintPreview("C1FlexSheet", scaleMode, new Thickness(96), int.MaxValue);
```