ComponentOne

FlexChart 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

Trademarks

The ComponentOne product name is a trademark and ComponentOne is a registered trademark of GrapeCity, Inc. All other trademarks used herein are the properties of their respective owners.

Warranty

ComponentOne warrants that the media on which the software is delivered is free from defects in material and workmanship, assuming normal use, for a period of 90 days from the date of purchase. If a defect occurs during this time, you may return the defective media to ComponentOne, along with a dated proof of purchase, and ComponentOne will replace it at no charge. After 90 days, you can obtain a replacement for the defective media by sending it and a check for $2 5 (to cover postage and handling) to ComponentOne.

Except for the express warranty of the original media on which the software is delivered is set forth here, ComponentOne makes no other warranties, express or implied. Every attempt has been made to ensure that the information contained in this manual is correct as of the time it was written. ComponentOne is not responsible for any errors or omissions. ComponentOne's liability is limited to the amount you paid for the product. ComponentOne is not liable for any special, consequential, or other damages for any reason.

Copying and Distribution

While you are welcome to make backup copies of the software for your own use and protection, you are not permitted to make copies for the use of anyone else. We put a lot of time and effort into creating this product, and we appreciate your support in seeing that it is used by licensed users only.

Table of Contents

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Overview

Modern looking, high performance **FlexChart, FlexPie, Sunburst, FlexRadar** and **TreeMap** charts for **WPF** come with powerful features, flexible data binding, and simple and easy-to-use API. In addition, they provide several basic to complex chart types for your data visualization needs.

Following sections help you gain insights into these controls and their advanced capabilities.

- FlexChart
- FlexPie
- Sunburst Chart
- FlexRadar
- TreeMap

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Getting Started with WPF Edition

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

Copyright © 2019 GrapeCity, Inc. All rights reserved.

FlexChart

FlexChart—a powerful data visualization control for Windows—lets you add feature-rich and visually appealing charts to your Windows Forms applications. The control empowers end-users to visualize data that resonates with their audiences.

The FlexChart control provides you with numerous 2D chart types, built-in tools for chart interactivity, and diversified formats for chart rendering.

Whether it is storytelling with data or interpreting complex data, FlexChart helps you accomplish everything seamlessly.

Below is a complete listing of the sections to get you started and acquainted with the FlexChart control:

Key Features

FlexChart for WPF is an impeccable data visualization component in terms of performance, presentation, and overall quality. The control offers the following features:

1. **Automatic Legend generation:** Just specify the name of the series, and the Legend is displayed automatically.
2. **Axis labels automatic rotation:** Let long axis labels get rotated automatically, thereby rendering a clean appearance.
3. **Axis Grouping:** Group the axis labels while working with any kind of data (categorical, numeric or date time) for better readability and analysis.
4. **Chart export:** You can export your application to different formats, such as SVG, JPG, and PNG.
5. **Direct X support:** The control supports Direct X rendering engine.
6. **Flexible data labels:** Set offset, border, and position for data labels. The control also provides various options to manage overlapping of data labels such as automatic arrangement, hiding overlapped data labels and, rotation.
7. **In-built selection support:** Click on the chart and select either a single data point or an entire data series.
8. **Interpolate nulls:** Handle null values in line and area charts effectively by using the InterpolateNulls property.
9. **Legend wrapping:** Let the Legend items appear in several rows and columns depending upon the available space.

Copyright © 2019 GrapeCity, Inc. All rights reserved.

10. **Multiple chart types:** Add as many series as you want in a single chart. Set the desired chart type at each series, and thus integrate multiple chart types in a single chart.
11. **Pre-defined and custom palettes:** Choose from a number of pre-defined palettes or apply a custom palette to the chart.
12. **Powerful and flexible data binding:** Specify data source either at the series level or the chart level, as per your requirements. You can combine multiple data sources in a single chart as well.
13. **Series toggling:** Toggle the visibility of a series in the plot as well as the legend by using the LegendToggle property.
14. **Simple to use:** It is simple to work with the FlexChart control due to its clear object model.
15. **Stacked or 100% stacked charts:** Make a chart stacked or 100% stacked by setting just one property.
16. **Support for categorical, numerical, and data or time axis:** Bind to different data types ranging from int, float, and string to DateTime.
17. **ToolTip customization:** Leverage powerful tooltip customization features.
18. **Flexible data labels:** Set offset, border, and position for data labels. The control also provides various options to manage overlapping of data labels such as automatic arrangement, hiding overlapped data labels and, rotation.

Feature Comparison

This topic provides you with comparison charts that help you compare features offered by FlexChart across different platforms and features of FlexChart for WPF with those of another chart.

- Comparing FlexCharts
  Comparison of FlexChart across three platforms- WPF, WinForms, and UWP.
- Comparing WPF Charts
  Comparison of FlexChart for WPF with another chart.

Comparing FlexCharts

Explore all of the features offered by FlexChart in WPF, WinForms, and UWP. You can download the matrix in PDF.

Chart Types

Chart Types	WPF	Win	UWP
Area	✓	✓	✓
StackedArea	✓	✓	✓
StackedArea100	✓	✓	✓
SplineArea	✓	✓	✓

Copyright © 2019 GrapeCity, Inc. All rights reserved.

StackedSplineArea	✓	✓	✓
StackedSplineArea100	✓	✓	✓
StepArea	✓	✓	✓
Bar	✓	✓	✓
StackedBar	✓	✓	✓
StackedBar100	✓	✓	✓
Bubble	✓	✓	✓
CandleStick	✓	✓	✓
Column	✓	✓	✓
Heat Map	✓	✓	
StackedColumn	✓	✓	✓
StackedColumn100	✓	✓	✓
Histogram	✓	✓	✓
RangedHistogram	✓	✓	✓
Floating Bar	✓	✓	✓
Stock/ HighLowOpenClose	✓	✓	✓
Line	✓	✓	✓
LineStacked	✓	✓	✓
LineStacked100	✓	✓	✓
Spline	✓	✓	✓
SplineStacked	✓	✓	✓
SplineStacked100	✓	✓	✓
LineSymbols	✓	✓	✓
StackedLineSymbols	✓	✓	✓
StackedLineSymbols100	✓	✓	✓
SplineSymbols	✓	✓	✓
SplineSymbolsStacked	✓	✓	✓
StackedSplineSymbols100	✓	✓	✓
Step	✓	✓	✓
StepSymbols	✓	✓	✓
Pareto	✓	✓	✓
Pie	✓	✓	✓
Doughnut	✓	✓	✓

Copyright © 2019 GrapeCity, Inc. All rights reserved.

PieExploded	✓	✓	✓
DoughnutExploded	✓	✓	✓
Point/ Scatter	✓	✓	✓
Radar	✓	✓	✓
Polar	✓	✓	✓
Box-and-Whisker	✓	✓	✓
ErrorBar	✓	✓	✓
Funnel	✓	✓	✓
Sunburst	✓	✓	✓
Waterfall	✓	✓	✓
2D	✓	✓	✓
Heikin-Ashi	*	*	*
LineBreak/ThreeLineBreak	*	*	*
Renko	*	*	*
Kagi	*	*	*
ColumnVolume	*	*	*
EquiVolume	*	*	*
CandleVolume	*	*	*
ArmsCandleVolume	*	*	*

* Available in FinancialChart

DataBinding

Data Binding	WPF	Win	UWP
Objects implementing IEnumerable	✓	✓	✓

Core Features

Core Features	WPF	Win	UWP
Handle Empty/ Null Data Points	✓	✓	✓
HitTest	✓	✓	✓
Annotations	✓	✓	✓
Render Modes	Native/Direct2D/Direct3D	Native/DirectX	Native/Direct3D
Trend Lines	✓	✓	✓
Coordinate	✓	✓	✓

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Conversion Methods			
Batch Updates	✓	✓	✓
Serialization Support		✓	
Sunburst Drilldown	✓	✓	✓
TreeMap Drilldown	✓	✓	✓

Look & Feel

Look & Feel	WPF	Win	UWP
Predefined Palettes	16	16	16
Custom Palette	✓	✓	✓
Background Color	✓	✓	✓
Background Image	✓	✓	✓
Background Gradient/ HatchStyles	✓	✓	✓
Border and Border Styles	✓	✓	✓

Chart Area

Chart Area	WPF	Win	UWP
Header	✓	✓	✓
Footer	✓	✓	✓
Header/Footer Borders	✓	✓	✓
Header/Footer Alignment	✓	✓	✓
Rotate ChartArea	✓	✓	✓

Plot Area

Plot Area	WPF	Win	UWP
Plot margins	✓		
Markers for PlotElements	Supported on FlexChart with LineSymbols/SplineSymbols and Scatter chart types	Supported on FlexChart with LineSymbols/SplineSymbols and Scatter chart types	Supported on FlexChart with LineSymbols/SplineSymbols and Scatter chart types
Markers Size	✓	✓	✓

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Markers: Border and Border styling	✓	✓	✓
Background Image/ Gradient/ HatchStyle for PlotElements	✓	✓	✓

Data Labels

Data Labels	WPF	Win	UWP
Offset	✓	✓	✓
ConnectingLines	✓	✓	✓
Borders and Border styling	✓	✓	✓
Styling	✓	✓	✓
Format String	✓	✓	✓
Custom Content	✓	✓	✓
Manage Overlapping	✓	✓	✓
Positions for Cartesian charts	Bottom/ Center/ Left/ None/ Right/ Top	Bottom/ Center/ Left/ None/ Right/ Top	Bottom/ Center/ Left/ None/ Right/ Top
Positions for Pie charts	Center/ Inside/ Outside/ None	Center/ Inside/ Outside/ None	Center/ Inside/ Outside/ None

Annotations

Annotations	Win	WPF	UWP
Pre-defined Shapes	✓	✓	✓
Position	✓	✓	✓
Attaching Annotations	✓	✓	✓
Offset	✓	✓	✓
Styling	✓	✓	✓
Tooltip	✓	✓	✓

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Customization	✓	✓	✓

Axes

Axes	WPF	Win	UWP
Axes: Primary X/Y	✓	✓	✓
Axes: Secondary X/Y	✓	✓	✓
Axes: Multiple Secondary X/Y	✓	✓	✓
Axis Label: Format strings	✓	✓	✓
Axis Label: Hide	✓	✓	✓
Axis Label: styling	✓	✓	✓
Axis Range (Min/Max) values	✓	✓	✓
Axis: Hide	✓	✓	✓
Axis: Logarithmic	✓	✓	✓
Axis: Reverse	✓	✓	✓
AxisLine Styling	✓	✓	✓
Labels: Alignment	✓	✓	✓
Labels: Angle	✓	✓	✓
Labels: Manage overlapping	✓	✓	✓
Major/ Minor GridLines	✓	✓	✓
Major/ Minor TickMarks	✓	✓	✓
Major/ Minor Units	✓	✓	✓
Title and Title styling	✓	✓	✓
Configure Origin	Any value	Any value	✓
TickMarks Position	Cross/ Inside/ Outside/ None	Cross/ Inside/ Outside/ None	Cross/ Inside/ Outside/ None
Position	Top/ Bottom/ Left/ Right/ Auto/ None	Top/ Bottom/ Left/ Right/ Auto/ None	Top/ Bottom/ Left/ Right/ Auto/ None

Series

Series	WPF	Win	UWP

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Multiple Series	✓	✓	✓
Data binding	✓	✓	✓
Chart types at series level	✓	✓	✓
Styling	✓	✓	✓
Visibility	Plot/ Legend/ Both Plot and Legend/ Hidden	Plot/ Legend/ Both Plot and Legend/ Hidden	Plot/ Legend/ Both Plot and Legend/ Hidden
Conditional Formatting	✓	✓	✓

Legends

Legends	WPF	Win	UWP
Title	✓	✓	✓
Title Style	✓	✓	✓
Toggle Series Visibility from legend	✓	✓	✓
Orientation	Auto/ Vertical/ Horizontal	Auto/ Vertical/ Horizontal	Auto/ Vertical/ Horizontal
Position	Left/ Top/ Right/ Bottom	Left/ Top/ Right/ Bottom	Left/ Top/ Right/ Bottom
Custom Legend Icon	✓	✓	✓
Legend Text Wrap	✓	✓	✓
Legend Grouping	✓	✓	✓

Marker Symbols

Marker Symbols	WPF	Win	UWP
Box	✓	✓	✓
Dot	✓	✓	✓

User Interactions

User Interactions	WPF	Win	UWP
Tooltips	✓	✓	✓
Series selection	✓	✓	✓
Point selection	✓	✓	✓
LineMarkers aka Crosshairs	✓	✓	✓
Range Selector	✓	✓	✓

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Zooming	✓	✓	✓
Scrolling	✓	✓	✓
Axis Scrollbar	✓	✓	✓

Tooltips

Tooltips	WPF	Win	UWP
Auto tooltips	✓	✓	✓
Custom content	✓	✓	✓
Show Delay	✓	✓	✓
Styling	✓	✓	✓
Tooltips for different chart elements			

Pie Charts

Pie Charts	WPF	Win	UWP
Exploded slices	✓	✓	✓
Inner Radius	✓	✓	✓
Starting Angle of first slice	✓	✓	✓

Exporting/Importing & Printing

Exporting/Importing & Printing	WPF	Win	UWP
Export to JPEG/ JPG	✓	✓	✓
Export to PNG	✓	✓	✓
Export to SVG		✓	
Export to BMP	✓		✓
Printing support	✓	✓	✓

Footprint

Footprint	WPF	Win	UWP
Assembly Size	183KB	229KB	218KB

Comparing WPF Charts

Explore all of the features offered by FlexChart and WPF Toolkit Chart. You can download the matrix in PDF.

Chart Types

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Features	FlexChart	WPF Toolkit Chart
Area	✓	✓
StackedArea	✓	
StackedArea100	✓	
SplineArea	✓	
StackedSplineArea	✓	
StackedSplineArea100	✓	
StepArea	✓	
Bar	✓	✓
StackedBar	✓	
Floating Bar	✓	
StackedBar100	✓	
Bubble	✓	✓
CandleStick	✓	
Column	✓	✓
Combination charts	✓	✓
StackedColumn	✓	
StackedColumn100	✓	
Stock/ HighLowOpenClose	✓	
Histogram	✓	
Pareto	✓	
Ranged Histogram	✓	
Line	✓	
LineStacked	✓	
LineStacked100	✓	
Spline	✓	
SplineStacked	✓	
SplineStacked100	✓	
Step	✓	✓
StepSymbols	✓	✓
LineSymbols	✓	✓
StackedLineSymbols	✓	
StackedLineSymbols100	✓	
SplineSymbols	✓	

Copyright © 2019 GrapeCity, Inc. All rights reserved.

SplineSymbolsStacked	✓	
StackedSplineSymbols100	✓	
Pie	✓	✓
Doughnut	✓	
PieExploded	✓	
DoughnutExploded	✓	
Point/ Scatter	✓	✓
Radar	✓	✓
Polar	✓	✓
Box-and-whisker	✓	✓
ErrorBar	✓	✓
Funnel	✓	✓
Sunburst	✓	
Waterfall	✓	
2D	✓	✓
Heikin-Ashi	*	
LineBreak/ThreeLineBreak	*	
Renko	*	
Kagi	*	
ColumnVolume	*	
EquiVolume	*	
CandleVolume	*	
ArmsCandleVolume	*	

Data Binding

Features	FlexChart	WPF Toolkit Chart
Objects implementing IEnumerable	✓	✓

Core Features

Features	FlexChart	WPF Toolkit Chart
Handle Empty/ Null Data Points	✓	
HitTest	✓	✓
Trendlines	✓	
Coordinate Conversion Methods	✓	
Annotations	✓	

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Render Modes	Default/Direct2D/Direct3D	
Batch Updates	✓	
Moving Average	✓	
Chart Optimization		
Load Animation		✓
Transition Animation		
Grouping and Aggregation		
LineMarkers	✓	✓
Multiple axes	✓	
Multiple plot areas	✓	

Chart Features

Features	FlexChart	MS Chart
Annotations	✓	✓
Axis binding	✓	✓
Customizable data labels	✓	✓
Customizable headers and footers	✓	✓

Data Manipulations

Features	FlexChart	MS Chart
Aggregation	With custom code	✓
Sorting	With custom code	✓
TopN	With custom code	✓

Look & Feel

Features	FlexChart	WPF Toolkit Chart
Themes		
Predefined Palettes	16	
Custom Palette	✓	
Background Color	✓	✓
Background Image	✓	✓
Background Gradient	✓	✓
Border and Border Styles	✓	✓

Chart Area

Features	FlexChart	WPF Toolkit Chart

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Header	✓	✓
Footer	✓	
Multiple Headers		
Header/ Footer Borders	✓	✓
Header/ Footer Alignment	✓	
Rotate ChartArea	✓	
Multiple Chart Areas		
Toggle Visibility		
PlotArea Collection		

Plot Area

Features	FlexChart	WPF Toolkit Chart
Plot margins	✓	
Markers for PlotElements	Supported on FlexChart with LineSymbols/ SplineSymbols and Scatter chart types	
Markers Size	✓	
Plot Areas: Multiple		
Plot Elements: Background Image/ Gradient/Color	✓	✓
Plot Elements: Border and Border styling		✓
LineMarkers/CrossHair		

Data Labels

Features	FlexChart	WPF Toolkit Chart
Offset	✓	
ConnectingLines	✓	
Borders and Border styling	✓	
Styling	✓	
Format String	✓	
Custom Content	✓	
Positions for Cartesian charts	Bottom/ Center/ Left/ None/ Right/ Top	
Manage Overlapping	✓	

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Positions for Pie charts	Center/ Inside/ Outside/ None	

Axis

Features	FlexChart	WPF Toolkit Chart
Axes: Primary X/Y	✓	✓
Axes: Secondary X/Y	✓	✓
Axes: Multiple Secondary X/Y	✓	✓
Axis Label: Format strings	✓	✓
Axis Label: Hide	✓	
Axis Label: styling	✓	
Axis Range (Min/Max) values	✓	✓
Axis: Hide	✓	✓
Axis: Logarithmic	✓	
Axis: Reverse	✓	
AxisLine Styling	✓	
Labels: Angle	✓	
Labels: Manage overlapping	✓	
Major/ Minor GridLines	✓	✓
Major/ Minor TickMarks	✓	✓
Major/ Minor Units	✓	Only MajorUnit
TickMarks Thickness		
TickMarks Styling		✓
Title and Title Styling	✓	✓
Configure Origin	Any value	
TickMarks Position	Cross/ Inside/ Outside/ None	
Position	Top/ Bottom/ Left/ Right/ Auto/ None	Auto/Bottom/Left/Right/Top
Axis Label: Custom		
ArrowHead styling		
Axis Label: Staggered		
ScaleBreaks		
Predefined DateTime Interval Types		✓
Sort Order for CategoryAxis		✓

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Series

Features	FlexChart	WPF Toolkit Chart
Multiple Series	✓	✓
Data binding	✓	✓
Chart types at series level	✓	
Styling	✓	✓
Visibility	Plot/ Legend/ Both Plot and Legend/ Hidden	✓
Conditional Formatting	✓	
X and Y value functions		
RoundedRadius for Bar Charts		
Stacked charts	✓	

Legends

Features	FlexChart	WPF Toolkit Chart
Title	✓	✓
Title Style	✓	
Toggle Series Visibility from legend	✓	
Orientation	Auto/ Vertical/ Horizontal	
Position	Left/ Top/ Right/ Bottom	
Custom Legend Items		✓
Items Ordering		
Multiple Legends		

Marker Symbols

Features	FlexChart	WPF Toolkit Chart
Box	✓	
Cross	✓	
DiagonalCross		
Dot		
Diamond		
Triangle		
Star4		
Star8		

Copyright © 2019 GrapeCity, Inc. All rights reserved.

User Interactions

Features	FlexChart	WPF Toolkit Chart
Tooltips	✓	✓
Series selection	✓	
Point selection	✓	✓
Zooming	✓	
Scrolling	✓	
Drilldown	With custom code	With custom code
Axis Scrollbar	✓	
Range Selector	✓	
Draggable Markers		
3D Chart Rotations		

Tooltips

Features	FlexChart	WPF Toolkit Chart
Auto tooltips	✓	✓
Custom content	✓	
Show Delay	✓	
Styling	✓	
Tooltips for different chart elements		

Pie Charts

Features	FlexChart	WPF Toolkit Chart
Exploded slices	✓	
Inner Radius	✓	
Starting Angle of first slice	✓	

Exporting, Importing & Printing

Features	FlexChart	WPF Toolkit Chart
Export to JPEG/ JPG	✓	
Export to PNG	✓	
Export to BMP	✓	
Export to GIF		
Export to TIFF		
Export to PDF		

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Export to SVG		
Export to EMF		
Export to EMfDual		
Export to EMfPlus		
Export to WMP		
Save/ Load to Binary files		
Save/ Load to memory streams		
Save/ Load to XML		
Printing	✓	✓

Footprint

Feature	FlexChart	WPF Toolkit Chart
Assembly Size	144KB	276KB

Pricing

Feature	FlexChart	MS Chart
Pricing	$	$

Quick Start

This quick start guides you through a step-by-step process of creating a simple **FlexChart** application and running the same in Visual Studio.

Go through the below-mentioned steps to walk through the FlexChart control quickly.

1. Adding FlexChart to the Project
2. Binding FlexChart to a Data Source
3. Running the Project

Step 1: Adding FlexChart to the Project

This step enables you to create a simple chart application using **FlexChart for WPF**. When you add the FlexChart control to your Visual Studio project, you would have a functional column chart with dummy data.

Perform the following steps:

1. Create a new **WPF Application** in Visual Studio.
    1. Select **File | New | Project**. The **New Project** dialog box appears.
    2. In the **New Project** dialog box, select a language in the left-hand pane, and then select **WPF Application** from the list of applications in the center pane.
    3. Give your application a **Name**, and then select **OK**.
2. Open the **MainWindow.xaml** file.
3. Place your cursor between the **<Grid> </Grid>** tags within either your Window or your UserControl, depending on the type of application you've created.

Copyright © 2019 GrapeCity, Inc. All rights reserved.

4. Locate the **FlexChart** control in Visual Studio's **ToolBox**. Double-click the control to add it to your application. The following reference is added to the project:
   C1.WPF.FlexChart.4.dll
   If the reference is not added, you need to add the same manually - right-click the **References** folder in the **Solution Explorer** and select **Add | New Reference**.

   The XAML markup resembles the following:

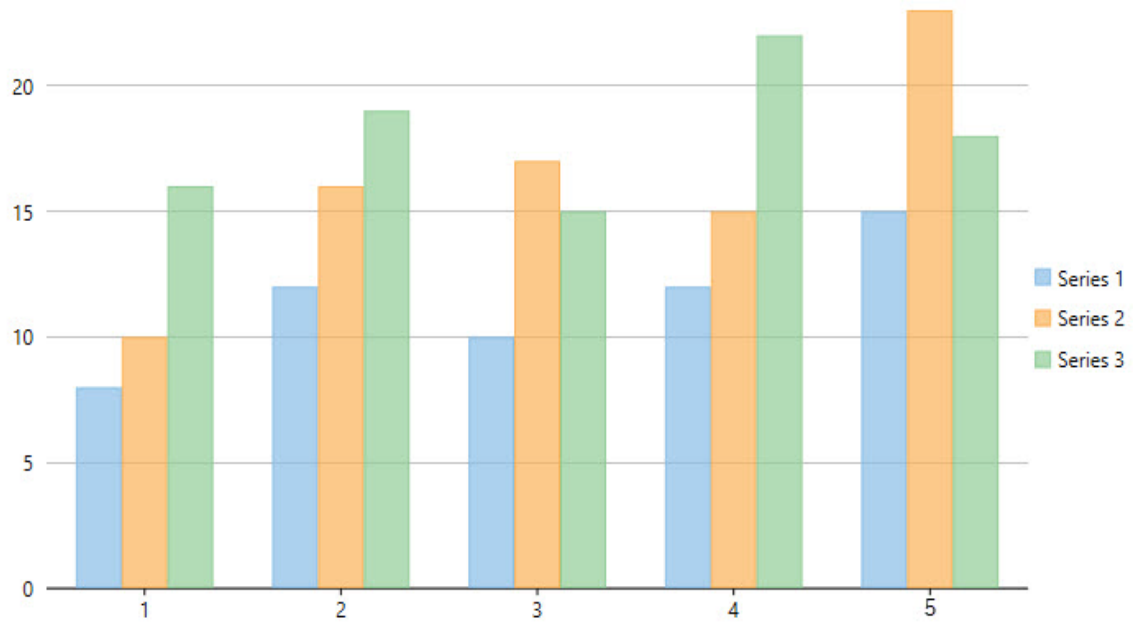
   - **XAML**

```
<Window
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:c1="http://schemas.componentone.com/winfx/2006/xaml"
        x:Class="QuickStart.MainWindow"
        Title="MainWindow" Height="886.465" Width="1182.397">
        <Grid Margin="46,0,0,12">

        <c1:C1FlexChart x:Name="flexChart"
                        HorizontalAlignment="Left"
                        Height="377"
                        VerticalAlignment="Top"
                        Width="702"
                        Margin="130,179,0,0">
            <c1:Series AxisX="{x:Null}"
                       AxisY="{x:Null}"
                       Binding="Y"
                       BindingX="X"
                       Chart="{x:Null}"
                       SeriesName="Series 1">
                <c1:Series.ItemsSource>
                    <PointCollection>1,8 2,12 3,10 4,12 5,15</PointCollection>
                </c1:Series.ItemsSource>
            </c1:Series>
            <c1:Series AxisX="{x:Null}"
                       AxisY="{x:Null}"
                       Binding="Y"
                       BindingX="X"
                       Chart="{x:Null}"
                       SeriesName="Series 2">
                <c1:Series.ItemsSource>
                    <PointCollection>1,10 2,16 3,17 4,15 5,23</PointCollection>
                </c1:Series.ItemsSource>
            </c1:Series>
            <c1:Series AxisX="{x:Null}"
                       AxisY="{x:Null}"
                       Binding="Y"
                       BindingX="X"
                       Chart="{x:Null}"
                       SeriesName="Series 3">
                <c1:Series.ItemsSource>
                    <PointCollection>1,16 2,19 3,15 4,22 5,18</PointCollection>
                </c1:Series.ItemsSource>
            </c1:Series>
        </c1:C1FlexChart>

    </Grid>
</Window>
```

5. Run the application.

Copyright © 2019 GrapeCity, Inc. All rights reserved.

The output appears as shown in figure below.



You have successfully created a WPF Application containing the FlexChart control.

Step 2: Binding FlexChart to a Data Source

This step allows you to bind **FlexChart** to a data source.

1. Create the data source as follows:
    1. Right-click the project and select **Add | Class.**
    2. Select **Class** from the list of templates, name it as **DataCreator.cs**, and click **Add**.
    3. Add the following code in **DataCreator.cs** class to generate the data.

        - **Visual Basic**

```
Public Class DataCreator Public Shared Function CreateData() As List(Of DataItem) Dim data
= New List(Of DataItem)() data.Add(New DataItem("UK", 5, 4)) data.Add(New DataItem("USA",
7, 3)) data.Add(New DataItem("Germany", 8, 5)) data.Add(New DataItem("Japan", 12, 8))
Return data End Function End Class Public Class DataItem Public Sub New(country__1 As
String, sales__2 As Integer, expenses__3 As Integer) Country = country__1 Sales = sales__2
Expenses = expenses__3 End Sub Public Property Country() As String Get Return m_Country End
Get Set m_Country = Value End Set End Property Private m_Country As String Public Property
Sales() As Integer Get Return m_Sales End Get Set m_Sales = Value End Set End Property
Private m_Sales As Integer Public Property Expenses() As Integer Get Return m_Expenses End
Get Set m_Expenses = Value End Set End Property Private m_Expenses As Integer End Class
```

        - **C#**

```
class DataCreator
{
    public static List<DataItem> CreateData()
    {
        var data = new List<DataItem>();
        data.Add(new DataItem("UK", 5, 4));
        data.Add(new DataItem("USA", 7, 3));
        data.Add(new DataItem("Germany", 8, 5));
        data.Add(new DataItem("Japan", 12, 8));
        return data;
    }
}

public class DataItem
{
    public DataItem(string country, int sales, int expenses)
    {
        Country = country;
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
            Sales = sales;
            Expenses = expenses;
        }

        public string Country { get; set; }
        public int Sales { get; set; }
        public int Expenses { get; set; }
    }
```

2. Bind the data to FlexChart as follows:

    1. Edit the **<Grid>** tag to the following markup to provide data to FlexChart:
        - **XAML**

```
<Grid>
    <c1:C1FlexChart x:Name="flexChart"
                    BindingX="Country"
                    ItemsSource="{Binding DataContext.Data}" Rendered="flexChart_Rendered">
        <c1:C1FlexChart.Series>
            <c1:Series SeriesName="Sales"
                       Binding="Sales"/>
            <c1:Series SeriesName="Expenses"
                       Binding="Expenses"/>
        </c1:C1FlexChart.Series>
```

> To specify the binding source, you need to add the **DataContext = "{Binding RelativeSource={RelativeSource Mode=Self}}"** markup in the **<Window>** tag of the **MainWindow.xaml** file.

    2. Switch to Code view (**MainWindow.xaml.cs or** MainWindow.xaml.vb). Add the following code in the MainWindow class to plot the data in the chart

        - **MainWindow.xaml.vb**

```
Private _data As List(Of DataItem)
Public Sub New()
    Me.InitializeComponent()
End Sub

Public ReadOnly Property Data() As List(Of DataItem)
    Get
        If _data Is Nothing Then
            _data = DataCreator.CreateData()
        End If

        Return _data
    End Get
End Property
```

        - **MainWindow.xaml.cs**

```
private List<DataItem> _data;
public MainWindow()
{
    InitializeComponent();

}

public List<DataItem> Data
{
    get
    {
        if (_data == null)
        {
            _data = DataCreator.CreateData();
        }

        return _data;
    }
}
```

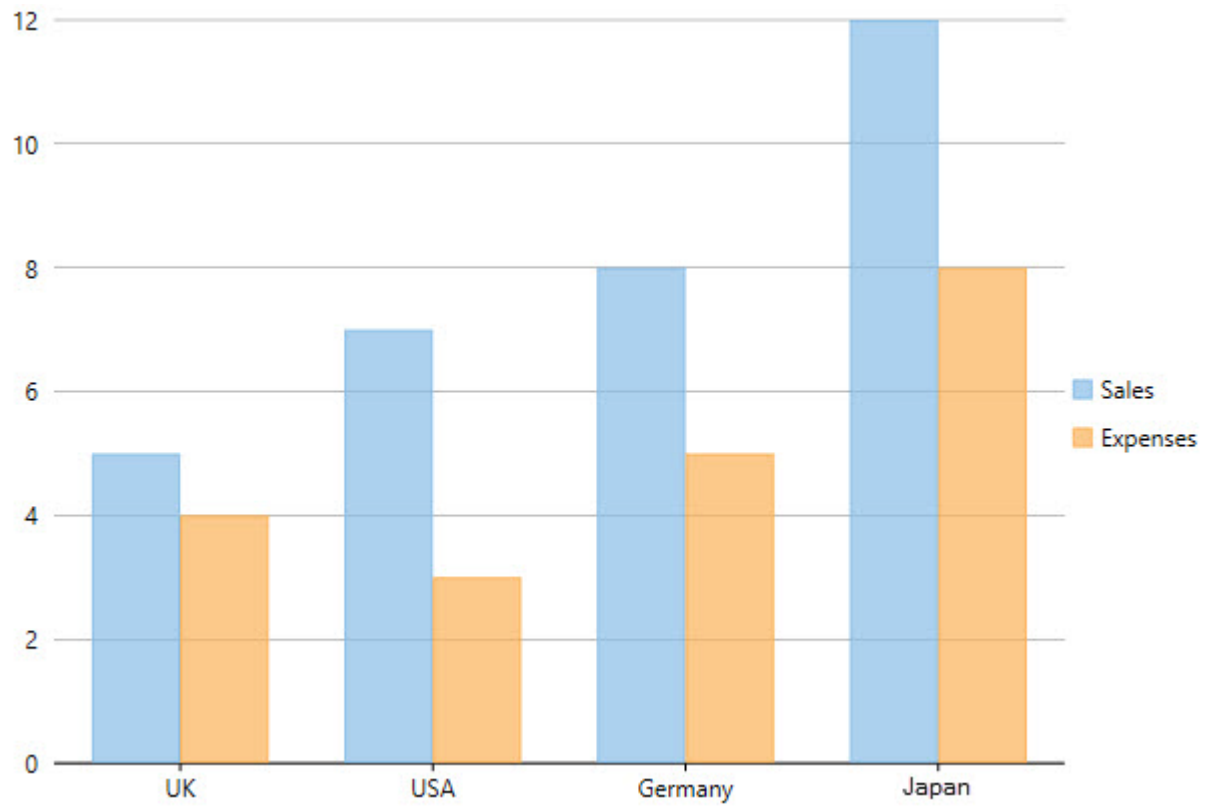
You have successfully bound FlexChart to the data source.

Step 3: Running the Project

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Now that you've created the WPF Application, the only thing left to do is run your application. Press F5 to run the application.

The output looks similar to the following image:



Understanding FlexChart

To get started with the **FlexChart** control, you need thorough understanding of all FlexChart fundamentals and FlexChart types.

This section, therefore, takes you through the basics of the control in an easy-to-understand manner.

Click the following links to find information on all FlexChart basics and types:

- FlexChart Fundamentals
- FlexChart Types

FlexChart Fundamentals

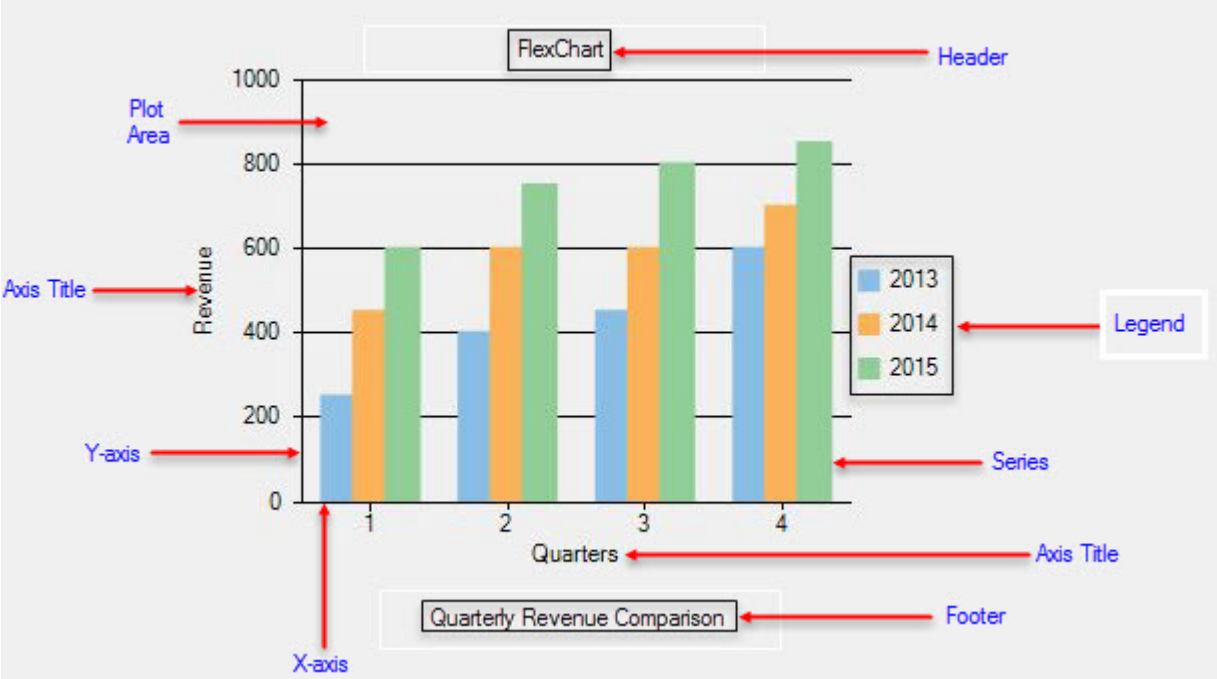
FlexChart consists of the following elements:

- Header and Footer
- Legend
- Axes
- Plot Area
- Series

The control has a rich object model that represents these elements in terms of objects and provides relevant properties for the same.

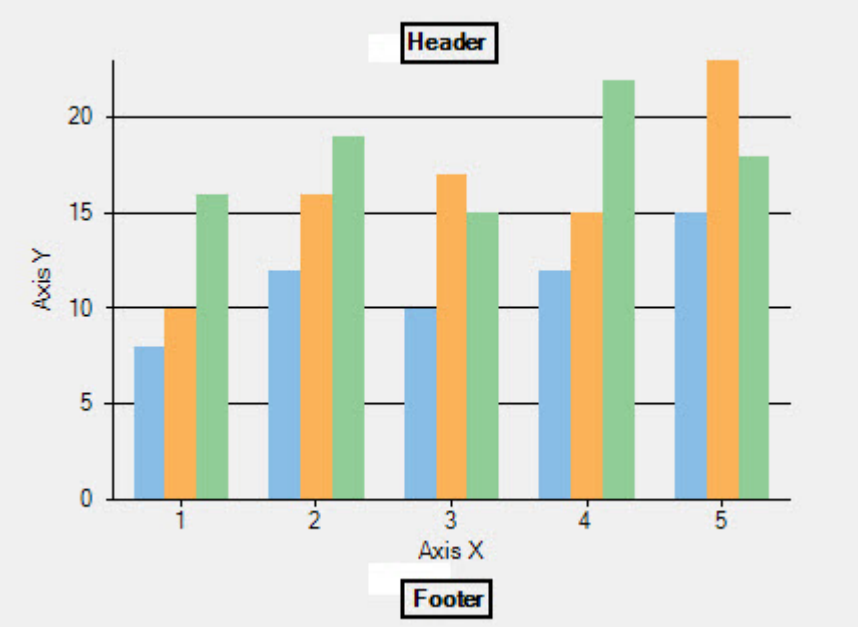
Copyright © 2019 GrapeCity, Inc. All rights reserved.

The following image displays the various elements:



Header and Footer

Header and Footer are used to display descriptive and relevant information with respect to the chart.

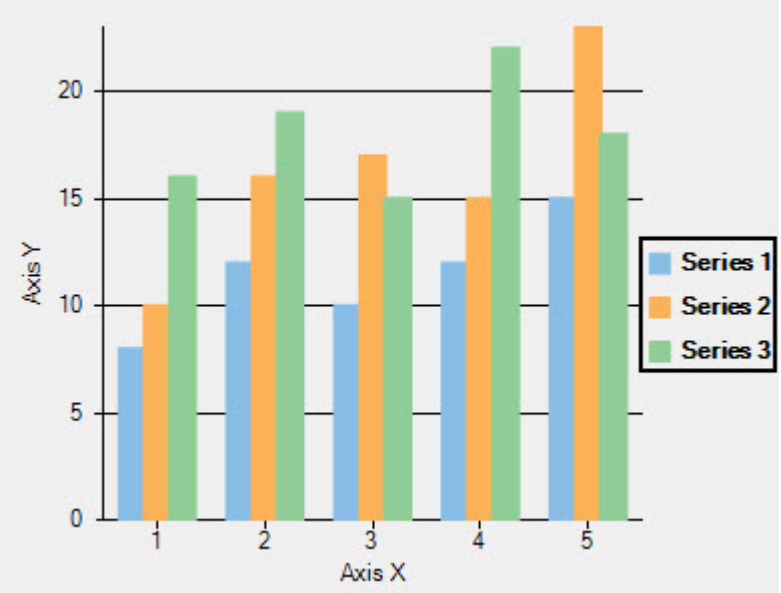


In FlexChart, these elements are set by using the Header and the Footer property. Both properties are the members of the FlexChartBase class.

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Legend

The Legend displays an entry for each data series in the chart. It represents the mapping between colors, symbols, and data series.



In FlexChart, the Legend is automatically generated based upon the series that have been added into the chart. However, you can use different properties to customize the Legend accordingly.

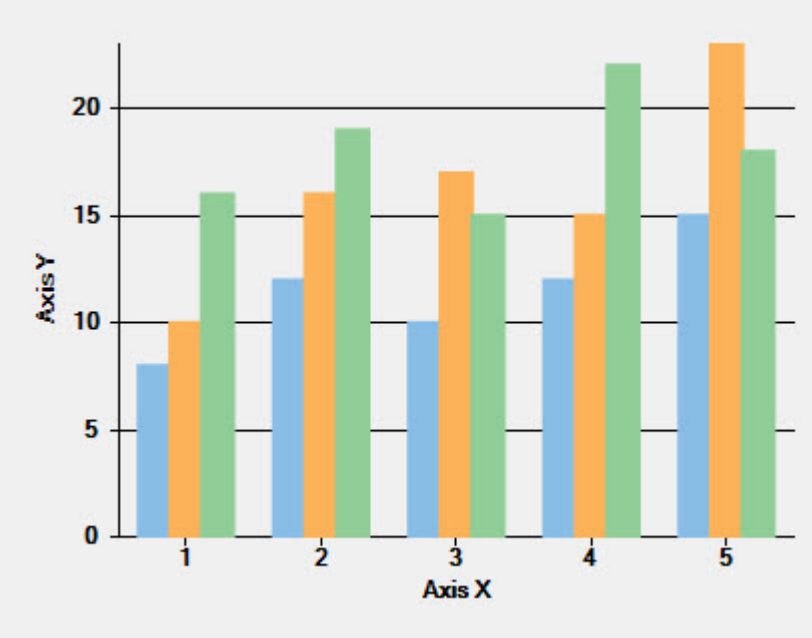
Property	Description
LegendPosition	Sets the position of the chart legend .
LegendStyle	Sets the style of the chart legend.
LegendToggle	Indicates whether clicking legend items toggles the series visibility in the chart.

For more details on Legend, refer to FlexChart Legend.

Axes

In a chart, there are two primary axes: X and Y. There are, of course, exceptions when you work with pie charts.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



In FlexChart, X-axis and Y-axis are represented by the AxisX and the AxisY property respectively. Both the properties return an Axis object comprising the properties given below:

Layout and Style Properties

Property	Description
AxisLine	Determines whether the axis line is visible.
AxisType	Contains the type of the axis.
Position	Sets the position of the axis.
Reversed	Reverses the direction of the axis.
Style	Sets the style of the axis.
Title	Sets the titles of the axis.

Axis Label Properties

Property	Description
Format	Sets the format string for the axis labels.
LabelAlignment	Sets the alignment of the axis labels.
LabelAngle	Set the rotation angle of the axis labels.
Labels	Determines whether the axis labels are visible.
MajorUnit	Sets the distance between axis labels.
OverlappingLabels	Indicates how to handle overlapping axis labels.

Axis Grouping Properties

Property	Description

Copyright © 2019 GrapeCity, Inc. All rights reserved.

GroupNames	Sets the group name for the axis labels.
GroupItemsPath	Sets the group name for the axis labels in hierarchical data.
GroupSeparator	Set the axis group separator.
GroupProvider	Sets the axis group provider.

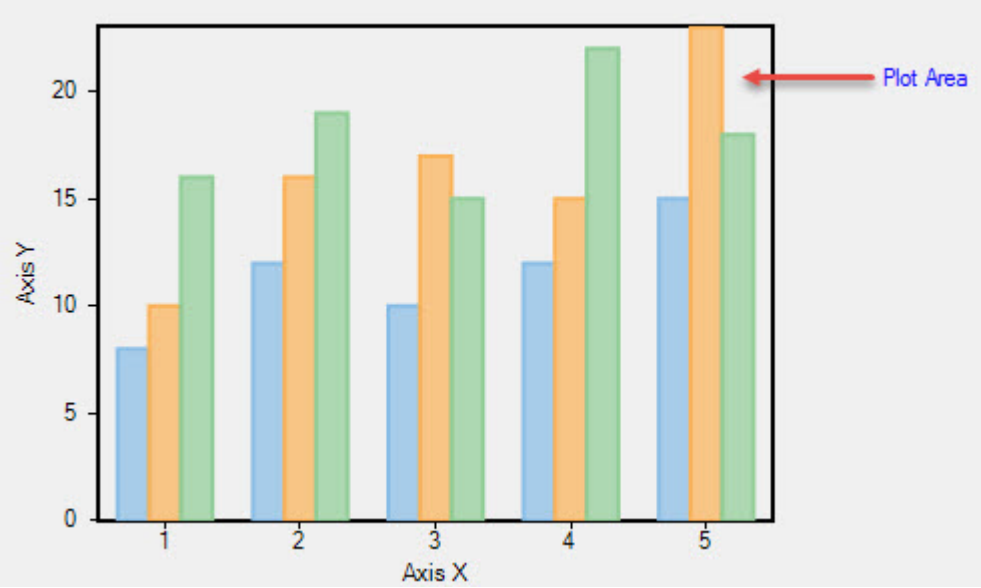
Scaling, Tick Mark, and Grid line Properties

Property	Description
MajorGrid	Determines whether the axis includes major grid lines.
MajorGridStyle	Contains properties to control the appearance of the major grid lines drawn perpendicular to the major tickmarks.
MinorGrid	Determines whether the axis includes minor grid lines.
MinorGridStyle	Contains properties to control the appearance of the minor grid lines drawn perpendicular to the minor tickmarks.
MajorTickMarks	Indicates the appearance of major tick marks.
MinorTickMarks	Indicates the appearance of minor tick marks.
Max	Sets the maximum value for the axis.
Min	Sets the minimum value for the axis.
Origin	Sets the value at which an axis crosses the perpendicular axis.

For more information on Axes, refer to FlexChart Axes.

Plot Area

The Plot Area contains data plotted against X-axis and Y-axis.



In FlexChart, the Plot Area can be customized by using the PlotStyle property, which returns the ChartStyle object containing the following properties:

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Property	Description
Fill	Sets the fill brush.
Stroke	Sets the stroke brush.
StrokeDashArray	Sets the stroke dash array.
StrokeThickness	Sets the stroke thickness.

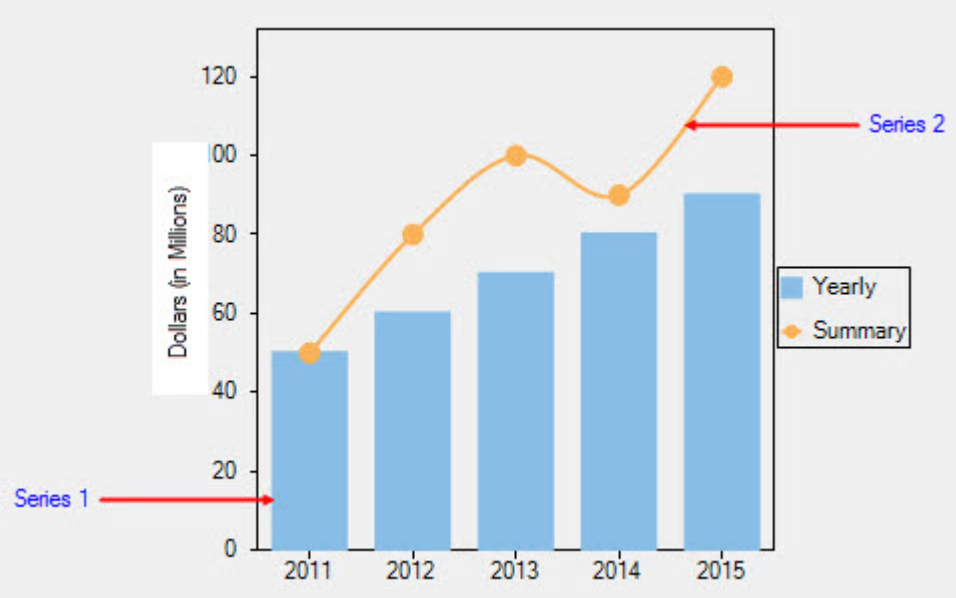
In addition, FlexChart allows creating multiple plot areas that increase data visibility by displaying a single series in a separate plot area.

For more information about multiple plot areas, see Multiple Plot Areas.

Series

Series are the groupings of the related points of data inside the Plot Area of the chart.

The following image illustrates data series in FlexChart:



You can access the series collection in FlexChart by using the Series property. And you can control the chart series by using the Series object comprising the following properties:

Property	Description
ActualBinding	Gets the actual binding
ActualBindingX	Gets the actual X-binding
ActualChart	Gets the actual chart.
ActualItemsSource	Gets the actual items source.
AxisX	Sets the series X-axis.
AxisY	Sets the series Y-axis.
Binding	Sets the name of the property that contains Y values for the series.

Copyright © 2019 GrapeCity, Inc. All rights reserved.

BindingX	Sets the name of the property that contains X values for the series.
ChartType	Sets the series chart type.
ItemsSource	Sets the collection of objects containing the series data.
MemberPaths	Sets the array of member bindings for the series.
SeriesName	Sets the series name.
Style	Sets the style of the series.
SymbolMarker	Sets the shape of the marker to be used for each data point in the series. This property applies to Scatter, LineSymbols, and SplineSymbols chart types only.
SymbolSize	Sets the size (in pixels) of the symbols used to render the series. This property also applies to Scatter, LineSymbols, and SplineSymbols chart types only.
Visibility	Sets the series visibility.

For more information on series, refer to FlexChart series.

FlexChart Types

FlexChart offers a comprehensive set of chart types to meet all data visualization requirements of your end-users.

Below is a full listing of all the chart types offered by the control. Based upon the chart type you would like to use in your application, click the corresponding link to avail key information on the same.

- Area Chart
- Bar Chart
- Bubble Chart
- Column Chart
- Financial Charts
- Floating Bar Chart
- Funnel Chart
- Gantt Chart
- Heat Map
- Histogram Chart
- Line Chart
- LineSymbols Chart
- Mixed Chart
- Pareto Chart
- RangedHistogram Chart
- Scatter Chart
- Spline Chart
- SplineArea Chart
- SplineSymbols Chart
- Step Chart

Area

The Area Chart depicts change in data over a period of time. It represents data series by connecting data points

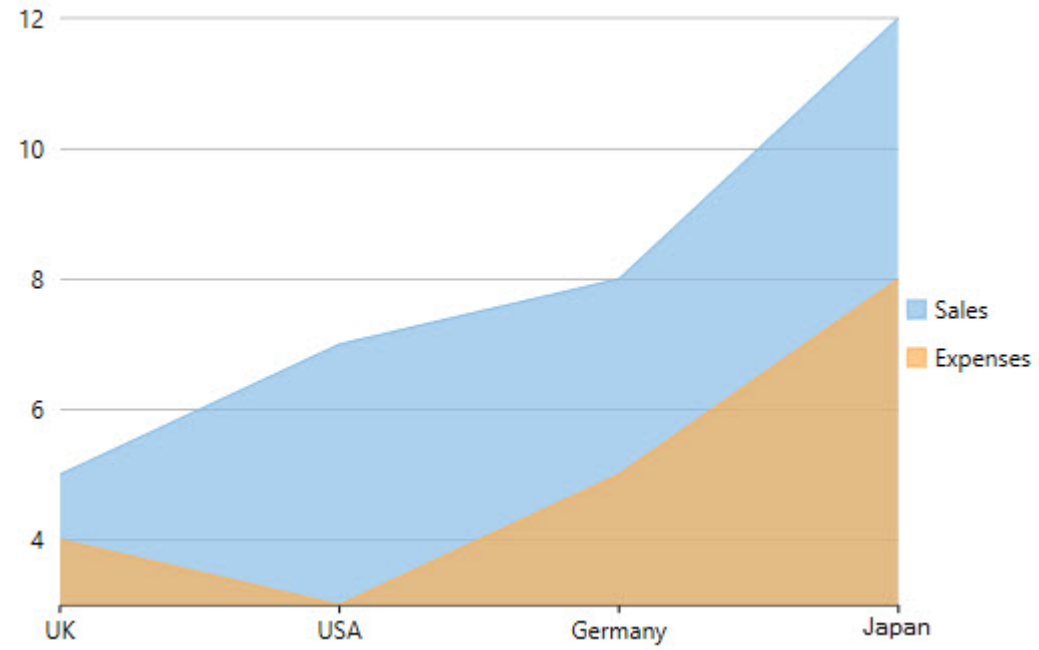
Copyright © 2019 GrapeCity, Inc. All rights reserved.

against Y-axis and filling the area between the series and X-axis. In addition, the chart displays data series in the same order in which they are added—back-to-front.

To create the Area Chart, you can set the ChartType property to Area in the Properties window or in XAML. Or you can also set the same programmatically.

In addition, you can set the Stacking property to Stacked or Stacked100pc to create the stacking Area Chart.

Area Chart



The following code implements the above-mentioned scenario:

XAML

```
<c1:C1FlexChart x:Name="flexChart"
                BindingX="Country"
                ItemsSource="{Binding DataContext.Data}"
                ChartType="Area"
                Margin="0,0,74,0">
    <c1:C1FlexChart.Series>
        <c1:Series SeriesName="Sales"
                   Binding="Sales"/>
        <c1:Series SeriesName="Expenses"
                   Binding="Expenses"/>
    </c1:C1FlexChart.Series>
</c1:C1FlexChart>
```

Code

C# copyCode

```
flexChart.ChartType = C1.Chart.ChartType.Area;
```

Bar

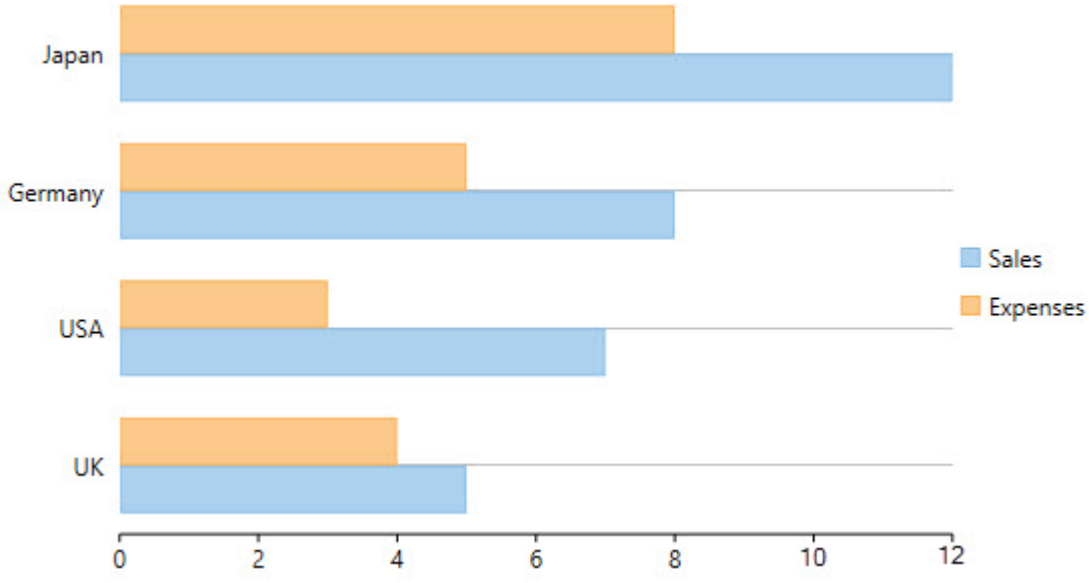
Copyright © 2019 GrapeCity, Inc. All rights reserved.

The Bar Chart compares values across various categories or displays variations in a data series over time. The chart displays horizontal bars for data series plotted against X-axis and arranges categories or items on Y-axis.

Set the ChartType property to Bar in the Properties window, in XAML, or in code to create the Bar Chart.

To create the stacking Bar Chart, you need to set the Stacking property either to Stacked or Stacked100pc.

Bar Chart



See the following code to implement the scenario:

XAML

```
<c1:C1FlexChart x:Name="flexChart"
                BindingX="Country"
                ItemsSource="{Binding DataContext.Data}"
                ChartType="Bar">
    <c1:C1FlexChart.Series>
        <c1:Series SeriesName="Sales"
                   Binding="Sales"/>
        <c1:Series SeriesName="Expenses"
                   Binding="Expenses"/>
    </c1:C1FlexChart.Series>
</c1:C1FlexChart>
```

Code

C# copyCode

```
flexChart.ChartType = C1.Chart.ChartType.Bar;
```

Bubble

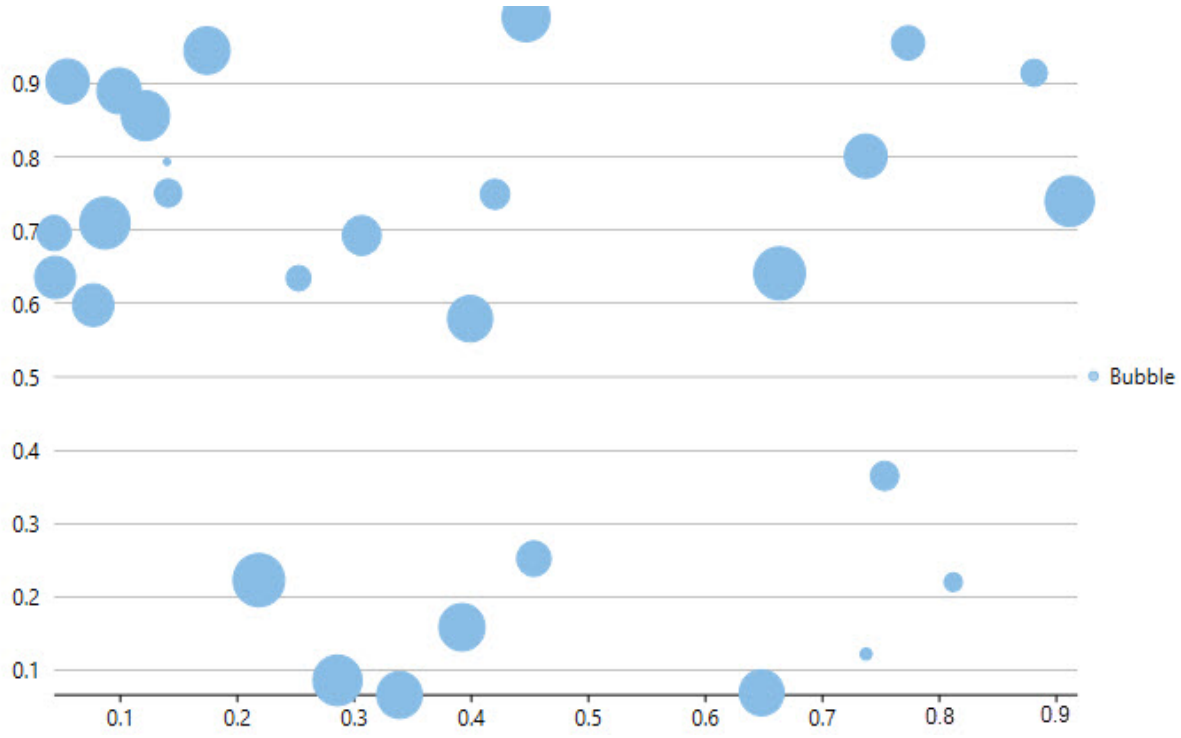
The Bubble Chart, which is basically a type of the Scatter Chart, is used for graphical representation of multi-dimensional data. It displays an additional data value at each point by changing its size. The chart type represents

Copyright © 2019 GrapeCity, Inc. All rights reserved.

data points in the form of bubbles (data markers) whose X and Y coordinates are determined by two data values and whose size indicates the value of a third variable.

To create the Bubble Chart, you need to set the ChartType property to Bubble either in the Properties window (at design-time) or code behind (at run-time). You can also set the same in XAML.

Bubble Chart



Below is the implementation in code:

XAML

```
<c1:C1FlexChart x:Name="flexChart"
                BindingX="X"
                ItemsSource="{Binding DataContext.Data}"
                ChartType="Bubble">
    <c1:C1FlexChart.Series>
        <c1:Series SeriesName="Bubble"
                   Binding="Y,Size"/>
    </c1:C1FlexChart.Series>
</c1:C1FlexChart>
```

Code

C#	copyCode

```
flexChart.ChartType = ChartType.Bubble;
```

Column

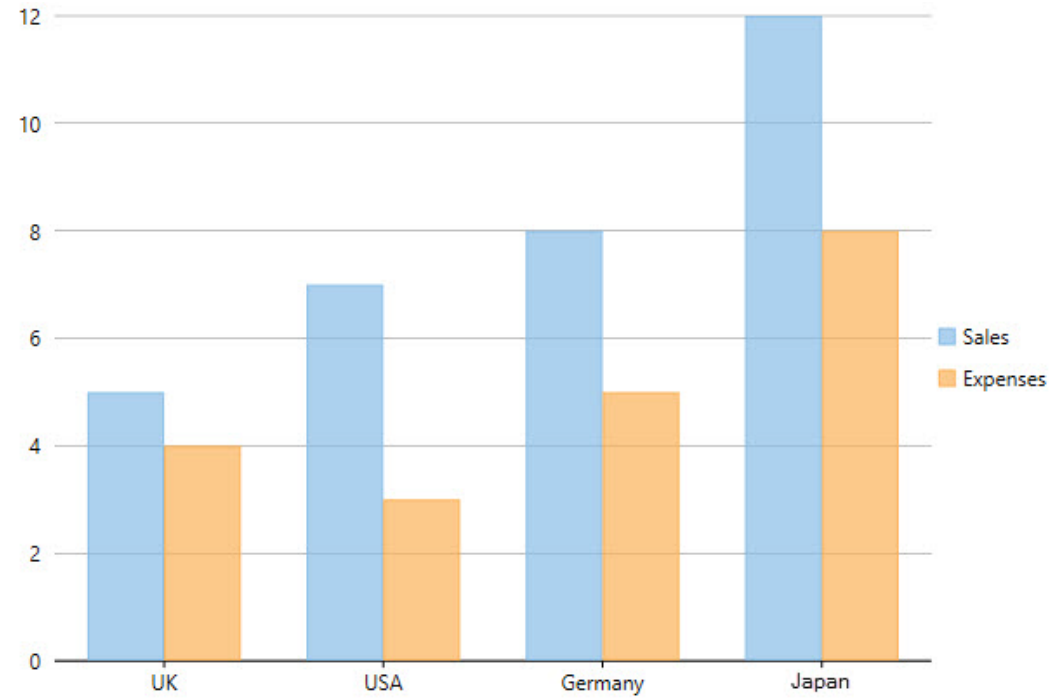
Copyright © 2019 GrapeCity, Inc. All rights reserved.

The Column Chart, just like the Bar Chart, represents variation in a data series over time or compares different items. It displays values of one or more items as vertical bars against Y-axis and arranges items or categories on X-axis.

You need to set the ChartType property to Column in the Property window, in the code behind, or in XAML to create the Column Chart.

Set the Stacking property to Stacked or Stacked100pc to create the stacking Column Chart.

Column Chart



The code below demonstrates the implementation:

XAML

```
<c1:C1FlexChart x:Name="flexChart"
                BindingX="Country"
                ItemsSource="{Binding DataContext.Data}"
                ChartType="Column">
    <c1:C1FlexChart.Series>
        <c1:Series SeriesName="Sales"
                   Binding="Sales"/>
        <c1:Series SeriesName="Expenses"
                   Binding="Expenses"/>
    </c1:C1FlexChart.Series>
</c1:C1FlexChart>
```

Code

C# copyCode

```
flexChart.ChartType = C1.Chart.ChartType.Column;
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Financial

Financial charts are used to represent fluctuation in market or stock prices; nonetheless, these charts can also be used to represent scientific data.

The **FlexChart** control supports two types of financial charts: Candle Chart and HighLowOpenClose Chart.

To use these chart types, you need to set the ChartType property either to Candlestick or to HighLowOpenClose at design-time or run-time or in XAML.

Following are the financial chart types:

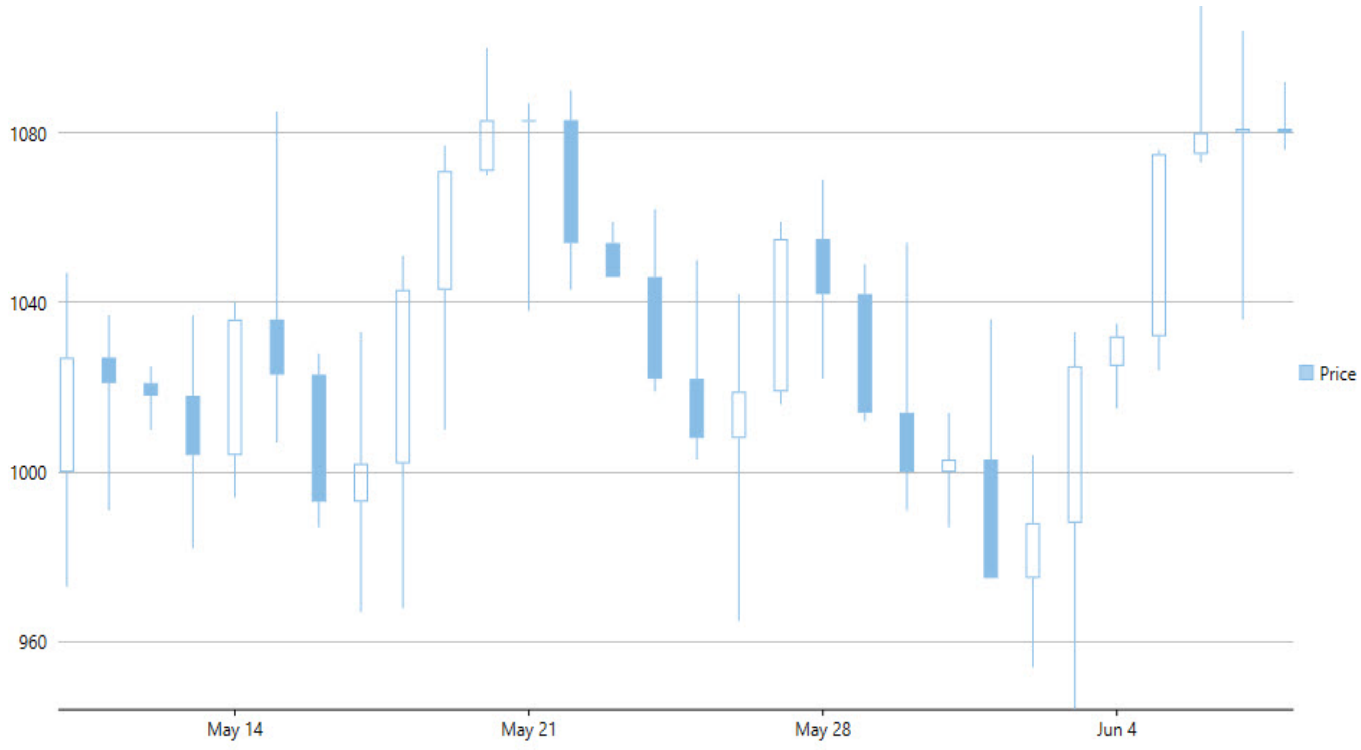
- Candle Chart
- HighLowOpenClose Chart

Candle

The Candle Chart integrates Bar and Line charts to depict a range of values over time. It consists of visual elements known as candles that are further comprised of three elements: body, wick, and tail.

- The body represents the opening and the closing value, while the wick and the tail represent the highest and the lowest value respectively.
- A hollow body indicates a rising stock price (the closing value is greater than the opening value).
- A filled body indicates a falling stock price (the opening value is greater than the closing value).

Candle Chart



Copyright © 2019 GrapeCity, Inc. All rights reserved.

The following code shows the implementation:

XAML

```
<c1:C1FlexChart x:Name="flexChart"
                BindingX="Time"
                ItemsSource="{Binding DataContext.Data}"
                ChartType="Candlestick">
    <c1:C1FlexChart.Series>
        <c1:Series SeriesName="Price"
                   Binding="High,Low,Open,Close"/>
    </c1:C1FlexChart.Series>
</c1:C1FlexChart>
```

Code

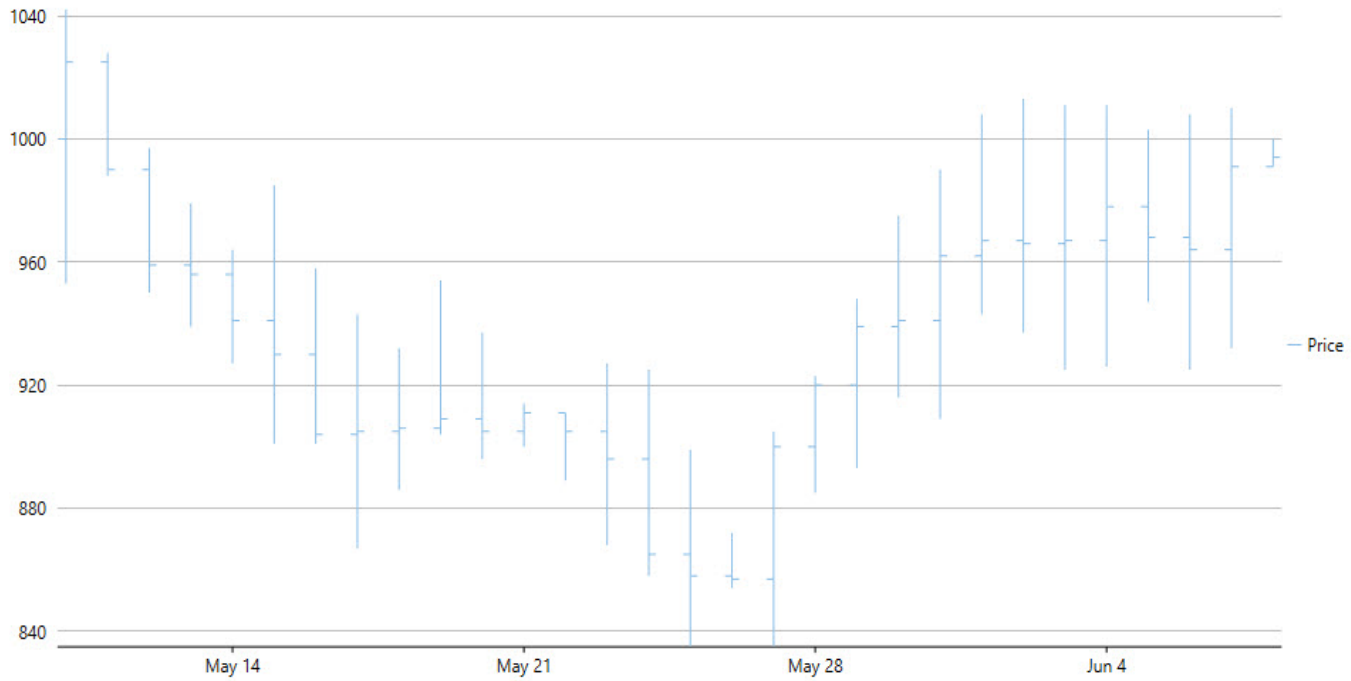
C#	copyCode

```
flexChart.ChartType = C1.Chart.ChartType.Candlestick;
```

HighLowOpenClose

The HighLowOpenClose Chart is generally used in stock analysis. The chart combines four independent values to supply high, low, open, and close data values for each data point in a series.

HighLowOpenClose Chart



See the following code for implementing the scenario:

XAML

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
<c1:C1FlexChart x:Name="flexChart"
                BindingX="Time"
                ItemsSource="{Binding DataContext.Data}"
                ChartType="HighLowOpenClose">
    <c1:C1FlexChart.Series>
        <c1:Series SeriesName="Price"
                   Binding="High,Low,Open,Close"/>
    </c1:C1FlexChart.Series>
</c1:C1FlexChart>
```

Code

C# copyCode

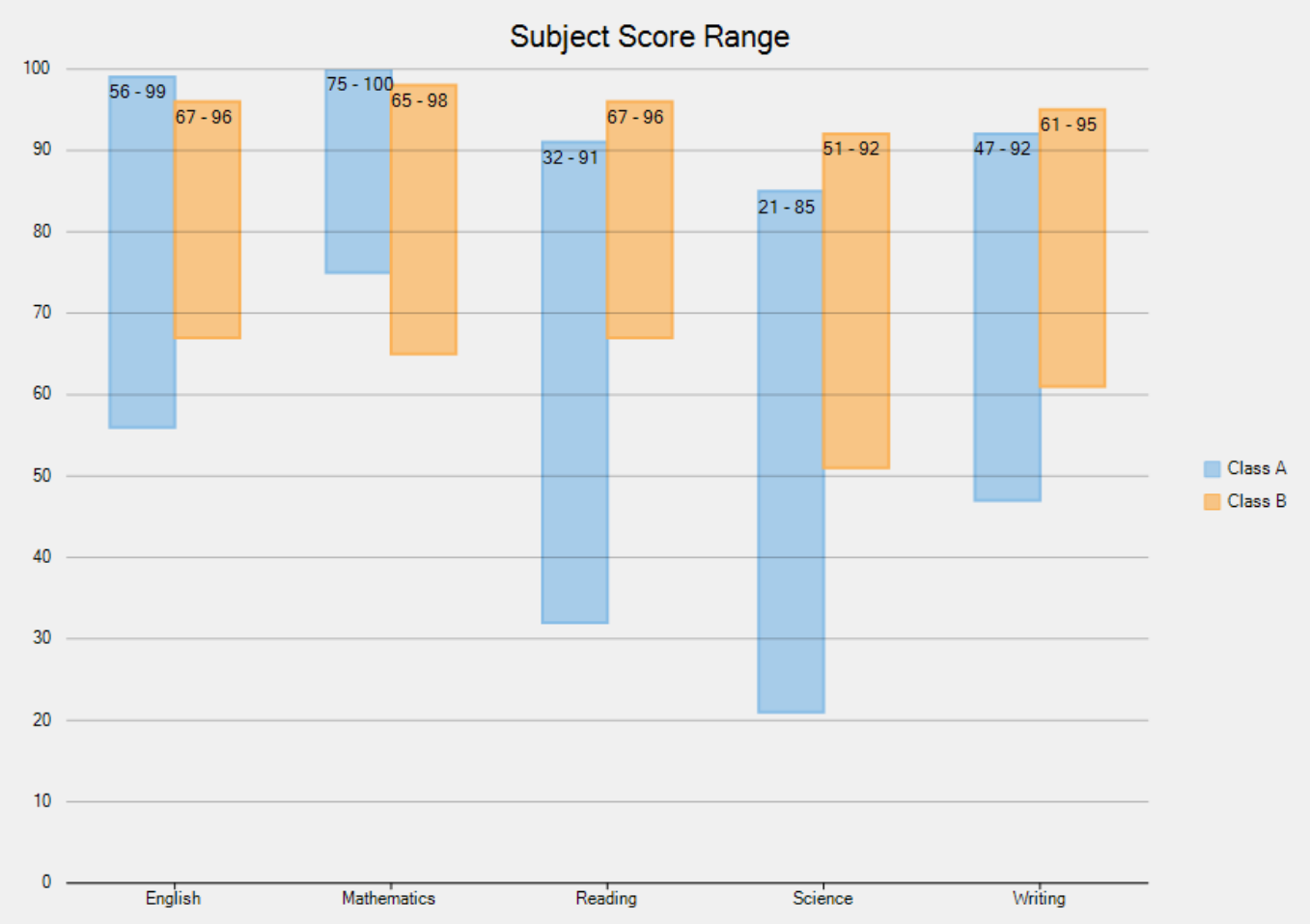
```
flexChart.ChartType = C1.Chart.ChartType.HighLowOpenClose;
```

Floating Bar

Floating bar chart provides an interesting way to represent data in charts. In this type of chart, a single or multiple bars apparently floats between a minimum and maximum value instead of being connected to the axis. It displays information as a range of data by plotting two Y-values(low and high) per data point. The Y-axis shows the values, and the X-axis shows the category they belong to. Floating bars can be useful to show highs and lows in a data set, such as daily high and low temperatures, stock prices, blood pressure readings, etc.

In FlexChart, Floating bar chart can be implemented using the Series class. To begin with, create a new Series object and specify its properties. Then, use the SymbolRendering event provided by the Series class to plot the data points on the chart.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



To implement Floating bar chart using the FlexChart control, see **FloatingBarChart** sample. The samples are available in the default installation folder - *Documents\ComponentOne Samples*

Funnel

A funnel chart allows you to visualize a linear process having connected stages. For instance, a sales process that tracks prospects across the stages, such as Sales Prospects, Qualified Prospects, Price Quotes, Negotiations, and Closed Sales.

In the process, each stage represents a proportion (percentage) of the total. Therefore, the chart takes the funnel shape with the first stage being the largest and each following stage smaller than the predecessor.

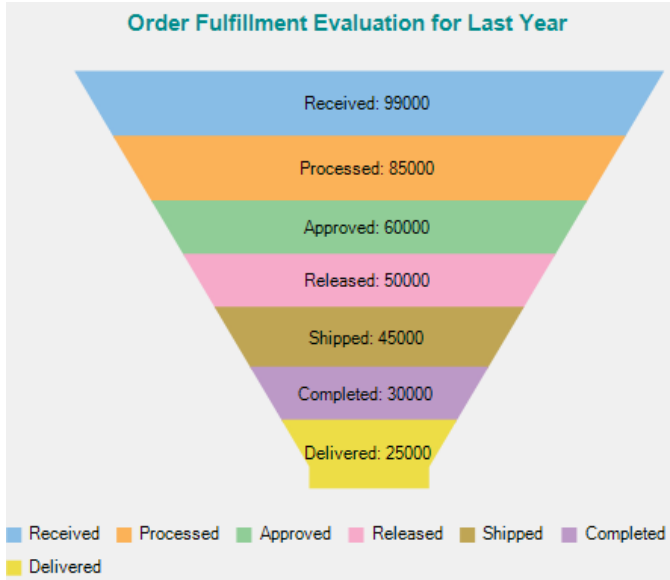
Funnel charts are useful in identifying potential problem areas in processes where it is noticeable at what stages and rate the values decrease.

FlexChart offers the Funnel chart in two forms, as follows.

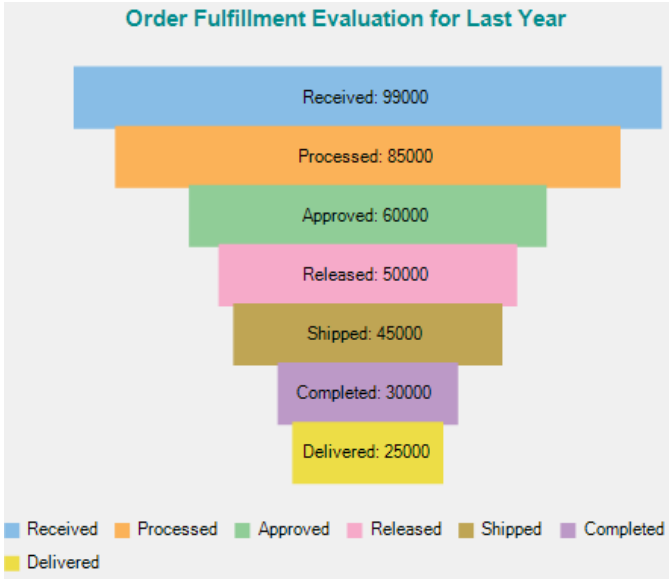
- **Trapezoid chart:** Contains a pair of parallel sides.
- **Stacked Bar chart**: Places related values on top of one another in the form of horizontal bars.

The following images show both Trapezoid and Stacked Bar charts displaying the number of orders across seven stages of an order fulfillment evaluation process.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



Trapezoid Chart



Stacked Bar Chart

In FlexChart, use the Funnel chart by setting the ChartType property to **Funnel** from the ChartType enum. Specify the type of the Funnel chart as either Trapezoid or Stacked Bar chart by setting the FunnelType property to Default or Rectangle from the FunnelChartType enum.

In addition, change the dimensions of the neck of the Funnel chart, when set as Trapezoid chart, by setting the FunnelNeckWidth and FunnelNeckHeight properties. These properties are available in the ChartOptions class accessible through the Options property of the C1FlexChart class.

The following code creates a class, **DataCreator** to create data containing values for the amount of orders across seven stages of an order fulfillment process.

- **Visual Basic**

```
Class DataCreator
    Public Shared Function CreateFunnelData() As List(Of DataItem)
        Dim data = New List(Of DataItem)()
        data.Add(New DataItem("Received", 99000))
        data.Add(New DataItem("Processed", 85000))
        data.Add(New DataItem("Approved", 60000))
        data.Add(New DataItem("Released", 50000))
        data.Add(New DataItem("Shipped", 45000))
        data.Add(New DataItem("Completed", 30000))
        data.Add(New DataItem("Delivered", 25000))
        Return data
    End Function
End Class

Public Class DataItem
    Public Sub New(order__1 As String, value__2 As Integer)
        Order = order__1
        Value = value__2
    End Sub

    Public Property Order() As String
        Get
            Return m_Order
        End Get
        Set
            m_Order = Value
        End Set
    End Property
    Private m_Order As String
    Public Property Value() As Integer
        Get
            Return m_Value
        End Get
        Set
            m_Value = Value
        End Set
    End Property
    Private m_Value As Integer
End Class
```

- **C#**

```
class DataCreator
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
{
    public static List<DataItem> CreateFunnelData()
    {
        var data = new List<DataItem>();
        data.Add(new DataItem("Received", 99000));
        data.Add(new DataItem("Processed", 85000));
        data.Add(new DataItem("Approved", 60000));
        data.Add(new DataItem("Released", 50000));
        data.Add(new DataItem("Shipped", 45000));
        data.Add(new DataItem("Completed", 30000));
        data.Add(new DataItem("Delivered", 25000));
        return data;
    }
}

public class DataItem
{
    public DataItem(string order, int value)
    {
        Order = order;
        Value = value;
    }

    public string Order { get; set; }
    public int Value { get; set; }
}
```

The following code sets the chart type as Funnel, specifies the dimensions of the Funnel neck, and sets Header, Legend, and Data Labels of the chart.

XAML

```
    <Window
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:FunnelChart"
        xmlns:c1="http://schemas.componentone.com/winfx/2006/xaml"
        x:Class="FunnelChart.MainWindow"
        mc:Ignorable="d"
        DataContext="{Binding RelativeSource={RelativeSource Mode=Self}}"
        Title="MainWindow" Height="705.284" Width="905.322">
    <Grid Margin="0,0,0,-117">

        <c1:C1FlexChart x:Name="flexChart"
                        BindingX ="Order"
                        ChartType="Funnel"
                        ItemsSource="{Binding Data}"
                        HorizontalAlignment="Left"
                        Height="471"
                        Margin="6,169,0,0"
                        VerticalAlignment="Top"
                        Width="792"
                        Header="Order Fulfillment Evaluation for Last Year"
                        HeaderAlignment="Center"
                        LegendPosition="Bottom">
            <c1:C1FlexChart.HeaderStyle>
                <c1:ChartStyle FontFamily="Arial"
                               FontSize="13"
                               FontWeight="Bold"
                               Stroke="DarkCyan"/>
            </c1:C1FlexChart.HeaderStyle>
            <c1:Series Binding="Value">
            </c1:Series>
            <c1:C1FlexChart.DataLabel>
                <c1:DataLabel Content="{}{Order}: {y}"
                              Position="Center"/>
            </c1:C1FlexChart.DataLabel>
            <c1:C1FlexChart.Options>
                <c1:ChartOptions FunnelType="Default"
                                 FunnelNeckHeight="0.05"
                                 FunnelNeckWidth="0.2"/>
            </c1:C1FlexChart.Options>
        </c1:C1FlexChart>
    </Grid>
</Window>
```

Code

Copyright © 2019 GrapeCity, Inc. All rights reserved.

MainWindow.xaml.vb copyCode

```
Partial Public Class MainWindow
    Inherits Window
    Private _data As List(Of DataItem)

    Public Sub New()
        InitializeComponent()
    End Sub

    Public ReadOnly Property Data() As List(Of DataItem)
        Get
            If _data Is Nothing Then
                _data = DataCreator.CreateFunnelData()
            End If

            Return _data
        End Get
    End Property
End Class
```

MainWindow.xaml.cs copyCode

```
public partial class MainWindow : Window
{
    private List<DataItem> _data;

    public MainWindow()
    {
        InitializeComponent();
    }

    public List<DataItem> Data
    {
        get
        {
            if (_data == null)
            {
                _data = DataCreator.CreateFunnelData();
            }

            return _data;
        }
    }
}
```

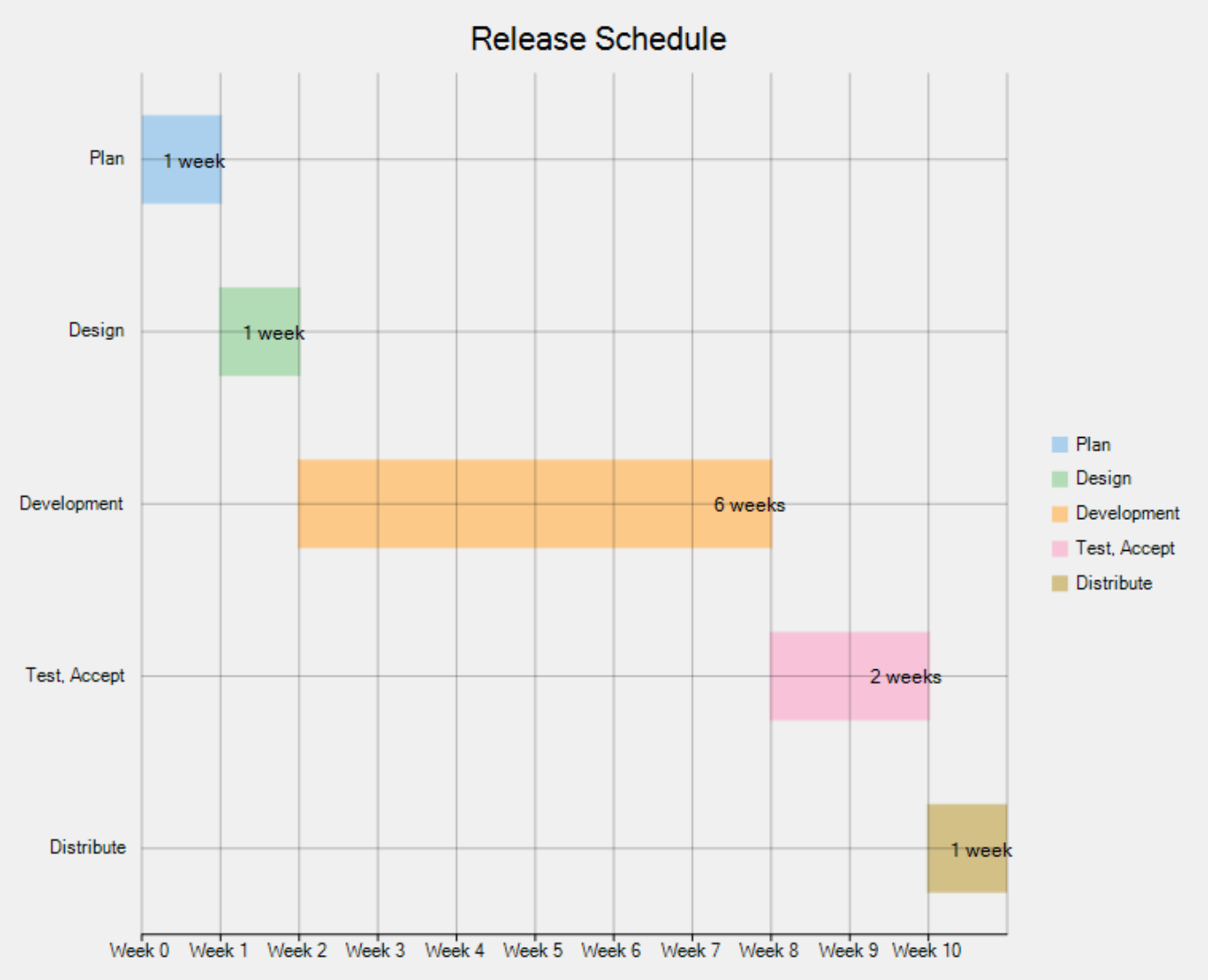
Gantt Chart

Gantt chart is a type of chart which is commonly used for planning and scheduling projects. It provides useful ways of showing activities that are displayed against time. The chart has a list of activities plotted on Y-axis while X-axis plots an appropriate time scale. Activities are represented using bars where position and length of the bars indicate the start time and duration of the activities respectively.

As primary purpose of a Gantt chart is planning and scheduling, it can be used for a range of projects and in various industries, such as construction, engineering, manufacturing, infrastructure, IT and more.

In FlexChart, Gantt chart can be implemented using the Series class. To begin with, create a new Series object and specify its properties. Then, use the SymbolRendering event provided by the Series class to plot activity bars on the chart and the LabelRendering event provided by the FlexChart class to display the labels.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



Heat Map

A heat map is a graphical representation of tabular data values through color coding and shading. Heat map helps in identifying patterns, areas of concentration, and data variance. The heat map data is represented by a two-dimensional matrix. You can use the heat map chart to plot data like event schedule, temperature records, weekly traffic intensity, etc.

A heat map chart can showcase different visual representations depending upon the type of data. Heat map chart can therefore be categorized into two types.

1. **Category based heat map**
2. **Value based heat map**

Category based heat map

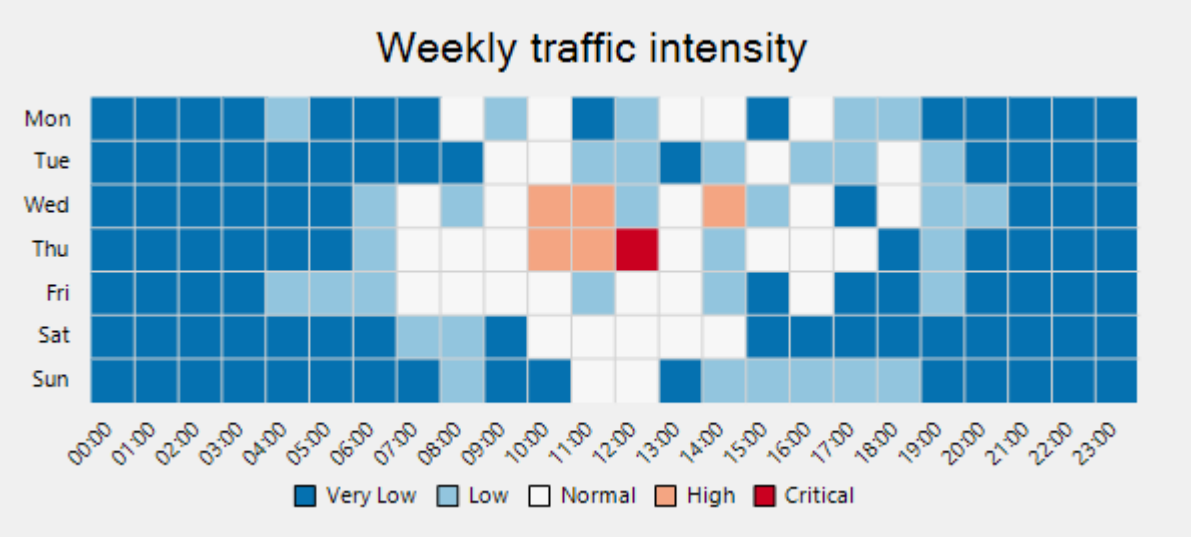
When you want to plot the data against intervals or specific numeric ranges, you can create a category based heat map. To implement category based heat map chart, you need to use the DiscreteColorScale class. It provides Intervals property that allows you to get or set the collection of intervals. Set the Min, Max, Color, and Name properties for each interval. So, the data points belonging to a particular interval are represented by the same color. The list of user-defined intervals is shown on the chart legend.

For example, if you use heat map chart to show the weekly traffic intensity of a city, you can use the

Copyright © 2019 GrapeCity, Inc. All rights reserved.

DiscreteColorScale class and add intervals such as very low, low, normal, and set its corresponding min, max, and color property.

The following image shows a category based heat map.



Use the following code snippet to create a category based heat map.

XAML

```
<UserControl x:Class="ExtendedFeatures.Samples.Loading"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             xmlns:c1="http://schemas.componentone.com/winfx/2006/xaml"
             mc:Ignorable="d"
             d:DesignHeight="300" d:DesignWidth="300">
    <Grid>
        <c1:C1FlexChart x:Name="chart" Loaded="ChartLoaded"
                        Header="Weekly traffic intensity"
                        ToolTipContent="{}{item:0.0}%" LegendPosition="Bottom">
            <c1:C1FlexChart.HeaderStyle>
                <c1:ChartStyle FontSize="20" />
            </c1:C1FlexChart.HeaderStyle>

            <c1:C1FlexChart.AxisX>
                <c1:Axis AxisLine="False" MajorGrid="False" MajorTickMarks="None"
LabelAngle="45"
                         MinorGrid="True">
                    <c1:Axis.MinorGridStyle>
                        <c1:ChartStyle Stroke="White" StrokeThickness="1" />
                    </c1:Axis.MinorGridStyle>
                </c1:Axis>
            </c1:C1FlexChart.AxisX>
            <c1:C1FlexChart.AxisY>
                <c1:Axis AxisLine="False" MajorGrid="False" MajorTickMarks="None"
Reversed="True"
                         MinorGrid="True">
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
                    <c1:Axis.MinorGridStyle>
                        <c1:ChartStyle Stroke="White" StrokeThickness="1"/>
                    </c1:Axis.MinorGridStyle>
                </c1:Axis>
            </c1:C1FlexChart.AxisY>
        </c1:C1FlexChart>
    </Grid>
</UserControl>
```

C#

```
public partial class Loading : UserControl
    {
        public Loading()
        {
            InitializeComponent();
        }
        private void ChartLoaded(object sender, RoutedEventArgs e)
        {
            chart.BeginUpdate();
            var scale = new DiscreteColorScale() { Intervals = new
List<DiscreteColorScale.Interval>() };
            scale.Intervals.Add(new DiscreteColorScale.Interval(0, 10,
Color.FromRgb(0x05,0x71,0xB0), "Very Low"));
            scale.Intervals.Add(new DiscreteColorScale.Interval(10, 25,
Color.FromRgb(0x92, 0xC5, 0xDE), "Low"));
            scale.Intervals.Add(new DiscreteColorScale.Interval(25, 75,
Color.FromRgb(0xD7, 0xD7, 0xD7), "Normal"));
            scale.Intervals.Add(new DiscreteColorScale.Interval(75, 90,
Color.FromRgb(0xF4, 0xA5, 0x82), "High"));
            scale.Intervals.Add(new DiscreteColorScale.Interval(90, 100,
Color.FromRgb(0xCA, 0x00, 0x20), "Critical"));

            var rnd = new Random();
            var data = new double[24, 7];
            for (var j = 0; j < 7; j++)
                for (var i = 0; i < 24; i++)
                    data[i, j] = 10 * Math.Exp(-(i - 12) * (i - 12) / (2 * 4.0 *
4.0))
                    / Math.Sqrt(2 * Math.PI * 4.0 * 4.0) * ((j == 5 || j == 6) ? 50 :
100)
                    * rnd.NextDouble();

            var hmap = new Heatmap();
            hmap.ItemsSource = data;
            hmap.ColorScale = scale;
            chart.Series.Add(hmap);

            var times = new string[24];
            for (var i = 0; i < 24; i++)
                times[i] = new DateTime(2000, 1, 1, i, 0, 0).ToShortTimeString();
            chart.AxisX.ItemsSource = times;
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

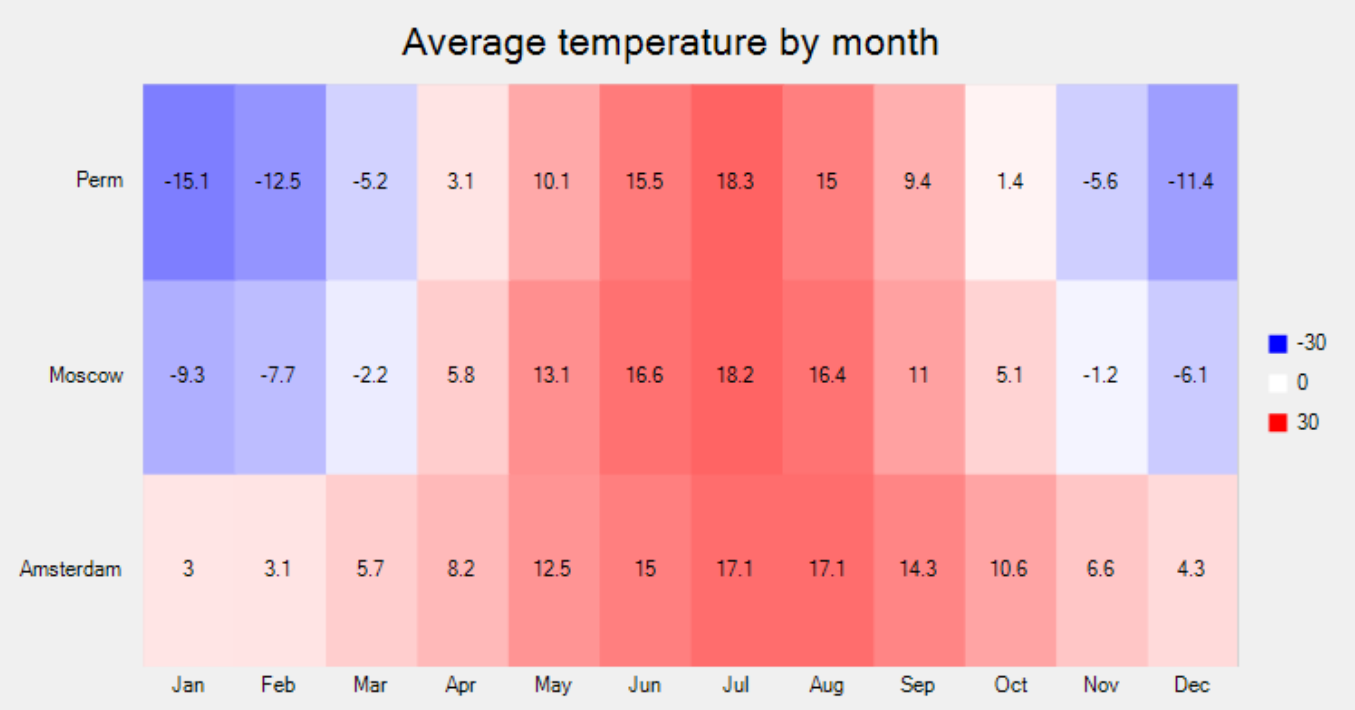
```
            chart.AxisY.ItemsSource = new string[] { "Mon", "Tue", "Wed", "Thu",
"Fri", "Sat", "Sun" };
            chart.EndUpdate();
        }
    }
```

Value based heat map

When you want to plot the raw data values without dividing them into intervals, you can create a value based heat map. To implement the value-based heat map, you need to use the GradientColorScale class. By default, a heat map is rendered with the chart legend. Each legend entry corresponds to a numeric value between the **Min** and **Max** values. The interval used to generate the numeric values for the legend entries is calculated by using the formula (|Min| + |Max|)/(n-1) where n refers to the number of colors specified in the Colors property.

For example, a simple custom palette containing red, white and blue color maps the values from -30 to 30 to shades of red, white, blue, where -30 is mapped to red and 30 to blue.

The following image shows a value based heat map.



Use the following code snippet to create a value based heat map.

XAML

```
<UserControl x:Class="ExtendedFeatures.Samples.Temperature"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             xmlns:c1="http://schemas.componentone.com/winfx/2006/xaml"
             mc:Ignorable="d"
             d:DesignHeight="300" d:DesignWidth="300" >
    <Grid>
        <c1:C1FlexChart x:Name="chart" Loaded="ChartLoaded" Rotated="True"
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
                        Header="Average temperature by month" ToolTipContent="">
            <c1:C1FlexChart.HeaderStyle>
                <c1:ChartStyle FontSize="20" />
            </c1:C1FlexChart.HeaderStyle>
            <c1:C1FlexChart.DataLabel>
                <c1:DataLabel Content="{}{item}" Position="Center" />
            </c1:C1FlexChart.DataLabel>
            <c1:C1FlexChart.AxisX>
                <c1:Axis AxisLine="False" MajorGrid="False" MajorTickMarks="None" />
            </c1:C1FlexChart.AxisX>
        </c1:C1FlexChart>
    </Grid>
</UserControl>
```

C#

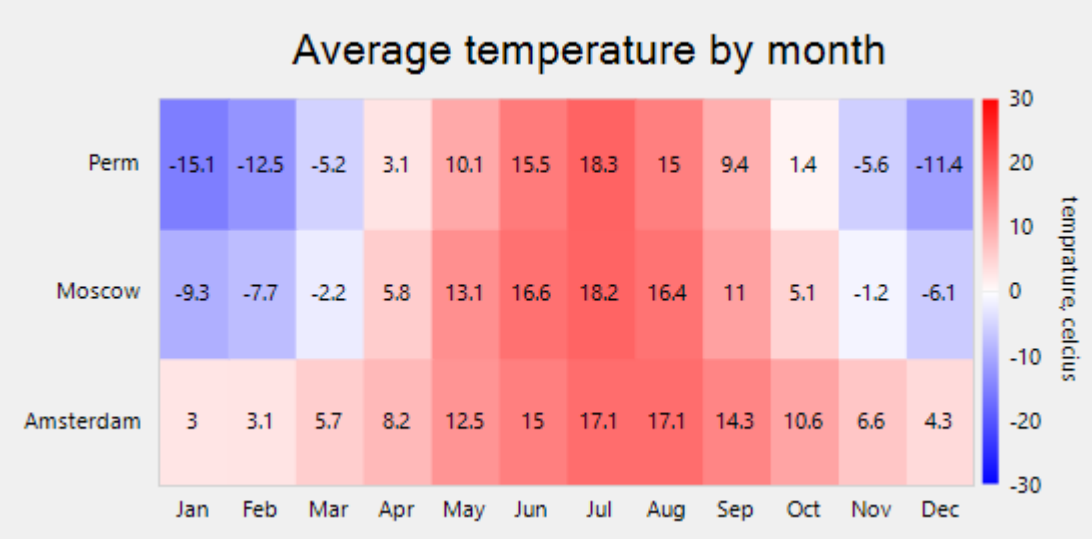
```
public partial class Temperature : UserControl
    {
        public Temperature()
        {
            InitializeComponent();
        }
        private void ChartLoaded(object sender, RoutedEventArgs e)
        {
            chart.BeginUpdate();
            chart.AxisY.MajorGrid = false;

            var scale = new GradientColorScale() { Min = -30, Max = 30 };
            scale.Colors = new List<Color> { Colors.Blue, Colors.White, Colors.Red };

            var hmap = new Heatmap();
            hmap.ItemsSource = new double[,] {
                {  3.0, 3.1, 5.7, 8.2, 12.5, 15.0, 17.1, 17.1, 14.3, 10.6, 6.6, 4.3
},
                { -9.3, -7.7, -2.2, 5.8, 13.1, 16.6, 18.2, 16.4, 11.0, 5.1, -1.2, -
6.1},
                 { -15.1, -12.5, -5.2, 3.1, 10.1, 15.5, 18.3, 15.0, 9.4, 1.4, -5.6, -
11.4},
                };
            hmap.ColorScale = scale;
            chart.Series.Add(hmap);
            chart.AxisX.ItemsSource = new string[] { "Jan", "Feb", "Mar", "Apr",
"May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };
            chart.AxisY.ItemsSource = new string[] { "Amsterdam", "Moscow", "Perm" };
            chart.EndUpdate();
        }
    }
```

The default chart legend can be replaced with a gradient chart legend by using the ColorAxis class. Gradient chart legend is a small bar of integrated colors obtained from the Colors definitions. Each color is for a specific numeric range and it integrates with the next color, creating a gradient legend. Every point on the gradient legend represents a distinctive color and value. Therefore, all the dissimilar values in the chart appear in distinctive colors as per the position on the gradient legend.

Copyright © 2019 GrapeCity, Inc. All rights reserved.

To implement heat map with a gradient legend, you need to provide an instance of ColorAxis class to the Axis property provided by the
GradientColorScale class. Moreover, you also need to specify the **Min**, **Max** and **Colors** property of GradientColorScale class to set up a gradient legend.

The following image shows a value based heat map with gradient legends.



Use the following code snippet to create a value based heat map with gradient legends.

XAML

```
<UserControl x:Class="ExtendedFeatures.Samples.Temperature"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             xmlns:c1="http://schemas.componentone.com/winfx/2006/xaml"
             mc:Ignorable="d"
             d:DesignHeight="300" d:DesignWidth="300" >
    <Grid>
        <c1:C1FlexChart x:Name="chart" Loaded="ChartLoaded" Rotated="True"
                        Header="Average temperature by month" ToolTipContent="">
            <c1:C1FlexChart.HeaderStyle>
                <c1:ChartStyle FontSize="20" />
            </c1:C1FlexChart.HeaderStyle>
            <c1:C1FlexChart.DataLabel>
                <c1:DataLabel Content="{}{item}" Position="Center" />
            </c1:C1FlexChart.DataLabel>
            <c1:C1FlexChart.AxisX>
                <c1:Axis AxisLine="False" MajorGrid="False" MajorTickMarks="None" />
            </c1:C1FlexChart.AxisX>
        </c1:C1FlexChart>
    </Grid>
</UserControl>
```

C#

```
public partial class Temperature : UserControl
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
    {
        public Temperature()
        {
            InitializeComponent();
        }
        private void ChartLoaded(object sender, RoutedEventArgs e)
        {
            chart.BeginUpdate();
            chart.AxisY.MajorGrid = false;

            var scale = new GradientColorScale() { Min = -30, Max = 30 };
            scale.Axis = scale.Axis = new C1.WPF.Chart.Extended.ColorAxis()
            { Position = Position.Right, Title = "temprature, celcius" };
            scale.Colors = new List<Color> { Colors.Blue, Colors.White, Colors.Red };

            var hmap = new Heatmap();
            hmap.ItemsSource = new double[,] {
                {  3.0, 3.1, 5.7, 8.2, 12.5, 15.0, 17.1, 17.1, 14.3, 10.6, 6.6, 4.3
},
                { -9.3, -7.7, -2.2, 5.8, 13.1, 16.6, 18.2, 16.4, 11.0, 5.1, -1.2, -
6.1},
                 { -15.1, -12.5, -5.2, 3.1, 10.1, 15.5, 18.3, 15.0, 9.4, 1.4, -5.6, -
11.4},
                };
            hmap.ColorScale = scale;
            chart.Series.Add(hmap);
            chart.AxisX.ItemsSource = new string[] { "Jan", "Feb", "Mar", "Apr",
"May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };
            chart.AxisY.ItemsSource = new string[] { "Amsterdam", "Moscow", "Perm" };
            chart.EndUpdate();
        }
    }
```

Histogram

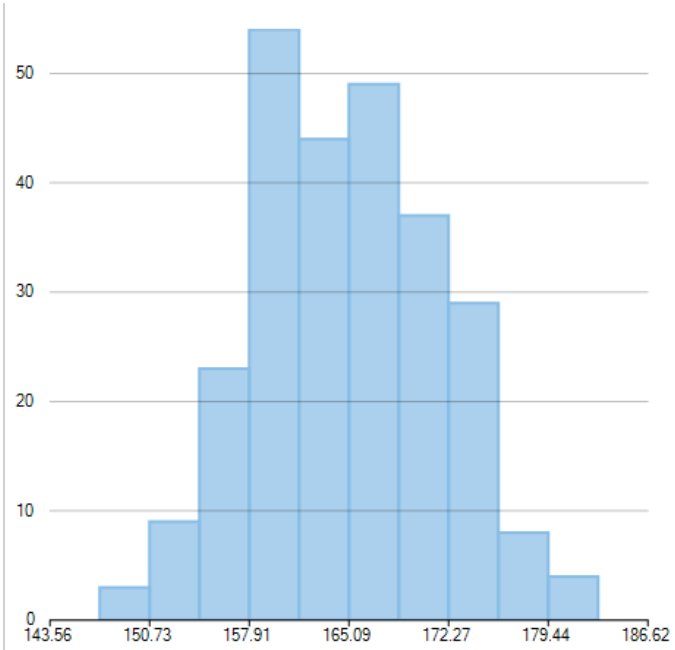
Histogram chart plots the frequency distribution of data against the defined class intervals or bins. These bins are created by dividing the raw data values into a series of consecutive and non-overlapping intervals. Based on the number of values falling in a particular bin, frequencies are then plotted as rectangular columns against continuous x-axis.

The following representations can created with the help of a histogram.

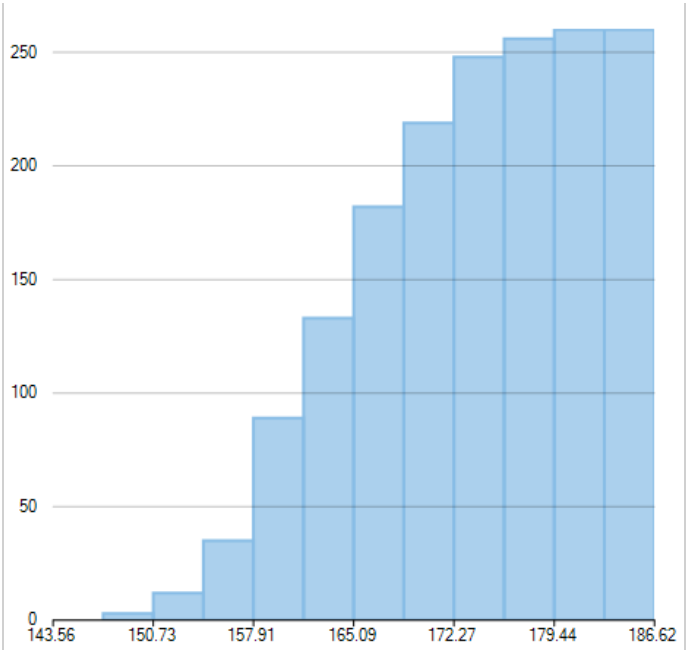
- **Frequency Polygon**
- **Gaussian Curve**

The following images show a histogram and a cumulative histogram created using FlexChart.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



Histogram



Cumulative Histogram

To create a histogram, you need to add the Histogram series and set the ChartType property to **Histogram**. Once you provide relevant data, FlexChart generates frequency distribution for the data and plots the same in histogram. The chart automatically calculates the intervals in which your data is grouped. However, if required, you can also specify the width of these intervals by setting the BinWidth property. Apart from this you can also create a cumulative histogram by setting the **CumulativeMode** property to **true**.

The following code snippet demonstrates how to generate Histogram chart for a particular data.

- **Xaml**

```
<Chart:C1FlexChart x:Name="flexChart"
                   ChartType="Histogram"
                   ItemsSource="{Binding DataContext.Data}"
                   Binding="Y"
                   BindingX="X">
    <Chart:C1FlexChart.AxisX>
        <Chart:Axis Format="0.00"></Chart:Axis>
    </Chart:C1FlexChart.AxisX>
    <Chart:Histogram x:Name="histogramSeries" SeriesName="Frequency"/>
</Chart:C1FlexChart>
```

Note that x-axis of Histogram chart can be shared by other chart series, which can be displayed together with the classic histogram series.

Frequency Polygon

A frequency polygon shows a frequency distribution representing the overall pattern in the data. It is a closed two-dimensional figure of straight line segments - created by joining the mid points of the top of the bars of a histogram.

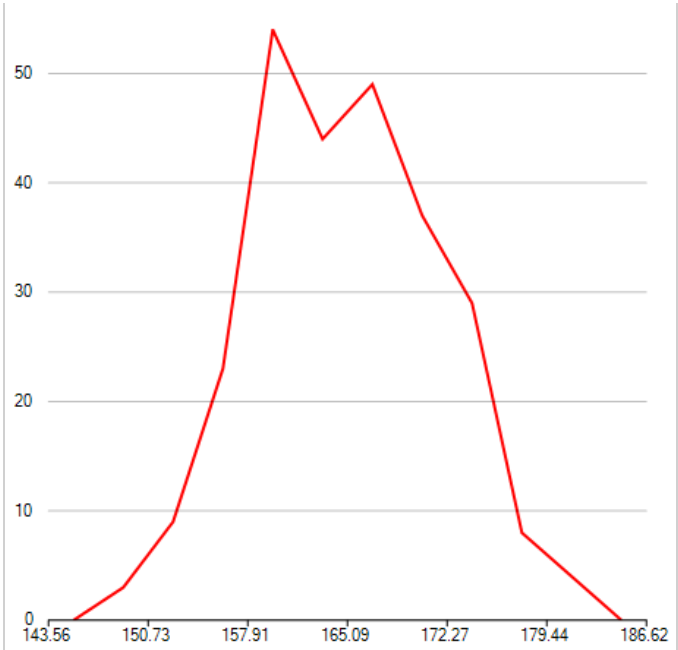
Use the following steps to create a frequency polygon using histogram chart.

1. Set the AppearanceType property to FrequencyPolygon. This property accepts value from the HistogramAppearance enumeration.
2. Set the style for frequency polygon using the FrequencyPolygonStyle property.

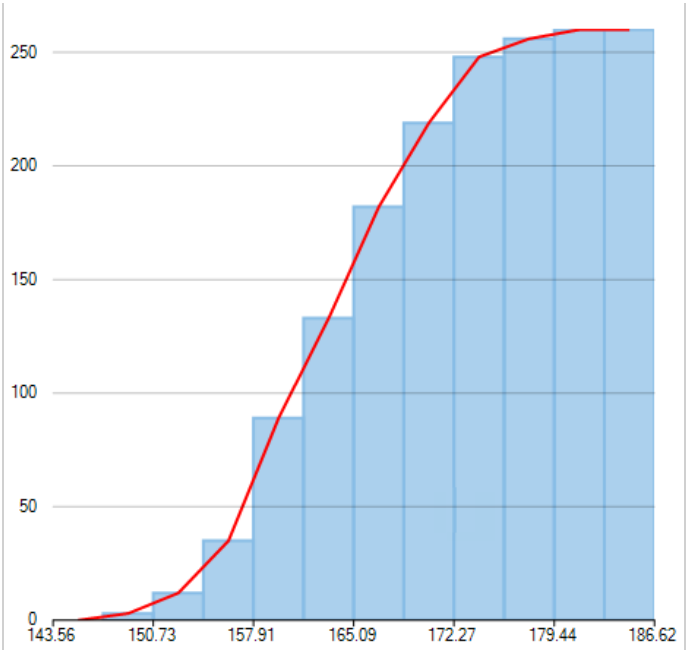
Moreover, you can also create a cumulative frequency polygon by setting the CumulativeMode property to **true**.

The following images show a frequency polygon and a cumulative frequency polygon created using FlexChart.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



Frequency Polygon



Cumulative Frequency Polygon

Use the following code snippet to create a frequency polygon.

hdfghdfg

In XAML

```
<c1:Histogram x:Name="histogramSeries" SeriesName="Frequency" CumulativeMode="True"
                 AppearanceType="FrequencyPolygon" />
<c1:Histogram.FrequencyPolygonStyle>
        <c1:ChartStyle Stroke="Red" StrokeThickness="2"/>
</c1:Histogram.FrequencyPolygonStyle>
```

In Code

```
histogramSeries.AppearanceType = HistogramAppearance.FrequencyPolygon;
histogramSeries.FrequencyPolygonStyle = new ChartStyle()
{Stroke = new SolidColorBrush(Color.FromRgb(255, 0, 0))};
// To create a cumulative frequency polygon
histogramSeries.CumulativeMode = true;
```

Gaussian Curve

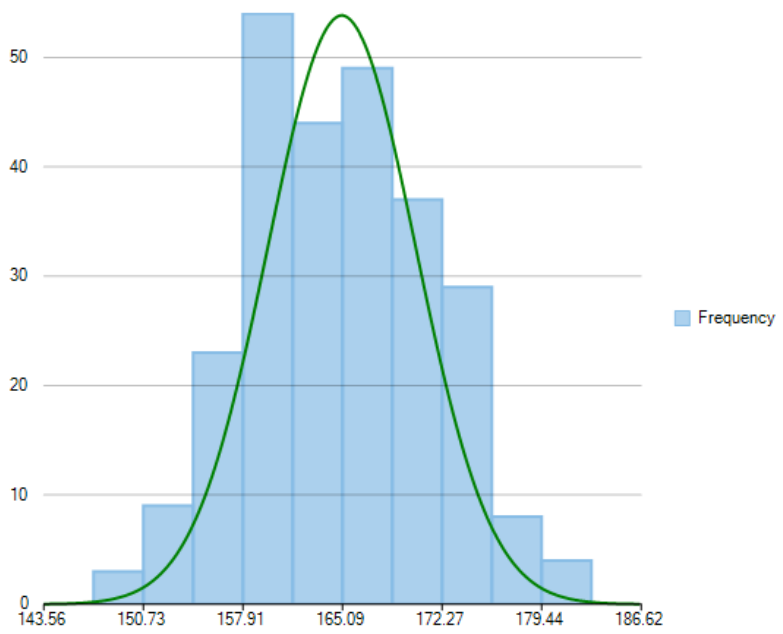
Gaussian curve is a bell shaped curve, also known as normal curve, which represents the probability distribution of a continuous random variable. It represents a unimodal distribution as it only has one peak. Moreover, it shows a symmetric distribution as fifty percent of the data set lies on the left side of the mean and fifty percent of the data lies on the right side of the mean.

Use the following steps to create a Gaussian curve using histogram chart.

1. Set the AppearanceType property to Histogram. This property accepts value from the HistogramAppearance enumeration.
2. Set the NormalCurve.Visible property to true to create a Gaussian curve.
3. Set the style for Gaussian curve using the NormalCurve.LineStyle property.

Following image illustrates a Gaussian curve created using FlexChart, which depicts probability distribution of scores obtained by students of a university in half yearly examinations.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



Use the following code snippet to create a Gaussian curve.

In XAML

```
<c1:Histogram x:Name="histogramSeries" SeriesName="Frequency"
                          AppearanceType="Histogram" />
<c1:Histogram.NormalCurve>
        <c1:NormalCurve>
        <c1:NormalCurve.LineStyle>
            <c1:ChartStyle Stroke="Green" StrokeThickness="2"/>
            </c1:NormalCurve.LineStyle>
    </c1:NormalCurve>
</c1:Histogram.NormalCurve>
```

In code

```
histogramSeries.AppearanceType = HistogramAppearance.Histogram;
histogramSeries.NormalCurve.Visible = true;
histogramSeries.NormalCurve.LineStyle = new ChartStyle() {Stroke = new SolidColorBrush(Color.FromRgb(0, 128, 0))};
```

Back to Top

Line

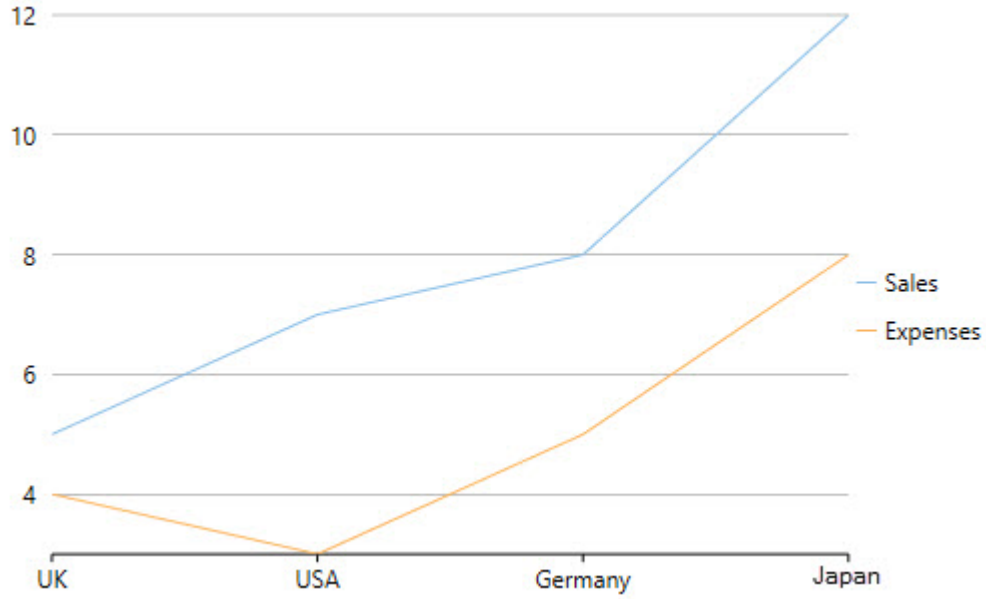
The Line Chart displays trends over a period of time by connecting different data points in a series with a straight line. It treats the input as categorical information that is evenly spaced along the X-axis.

You can create the Line Chart by setting the ChartType property to Line in XAML, at design-time, or using code.

To create the stacking Line Chart, you need to set the Stacking property to Stacked or Stacked100pc.

Line Chart

Copyright © 2019 GrapeCity, Inc. All rights reserved.



Below is the code that implements the aforementioned example:

XAML

```
<c1:C1FlexChart x:Name="flexChart"
                BindingX="Country"
                ItemsSource="{Binding DataContext.Data}"
                ChartType="Line">
    <c1:C1FlexChart.Series>
        <c1:Series SeriesName="Sales"
                   Binding="Sales"/>
        <c1:Series SeriesName="Expenses"
                   Binding="Expenses"/>
    </c1:C1FlexChart.Series>
</c1:C1FlexChart>
```

Code

C# copyCode

```
flexChart.ChartType = C1.Chart.ChartType.Line;
```

LineSymbols

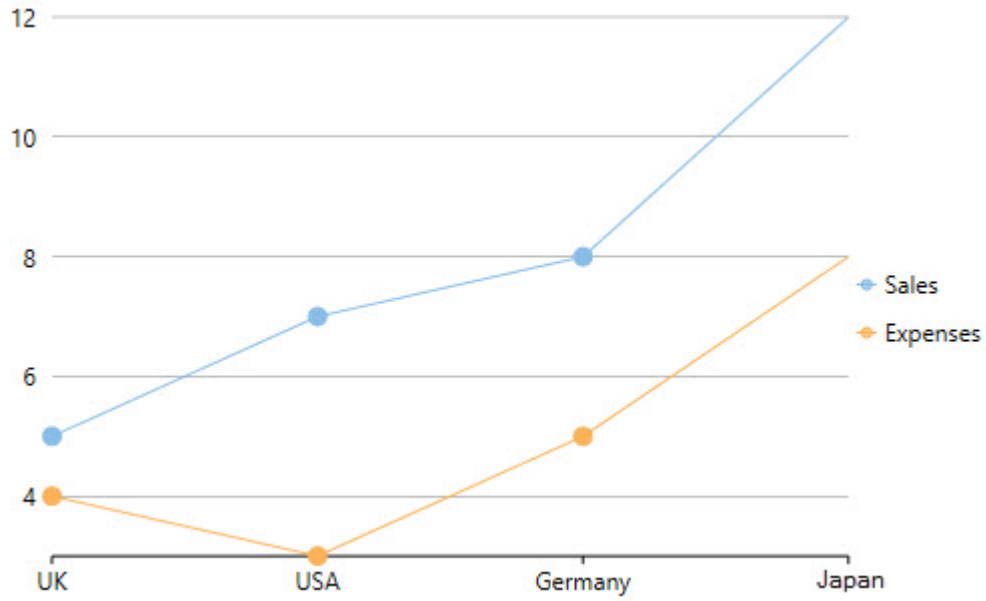
The LineSymbols Chart is a combination of the Line Chart and the Scatter Chart. The chart displays trends in data at equal intervals and visualizes relationship between two variables related to the same event. It plots data points by using symbols and connects the data points by using straight lines.

You need to set the ChartType property to LineSymbols either at design-time, in XAML, or at run-time to create the LineSymbols Chart.

You can set the Stacking property to Stacked or Stacked100pc to create the stacking LineSymbols Chart.

LineSymbols Chart

Copyright © 2019 GrapeCity, Inc. All rights reserved.



Below is the implementation in code:

XAML

```
<c1:C1FlexChart x:Name="flexChart"
                BindingX="Country"
                ItemsSource="{Binding DataContext.Data}"
                ChartType="LineSymbols">
    <c1:C1FlexChart.Series>
        <c1:Series SeriesName="Sales"
                   Binding="Sales"/>
        <c1:Series SeriesName="Expenses"
                   Binding="Expenses"/>
    </c1:C1FlexChart.Series>
</c1:C1FlexChart>
```

Code

C# copyCode

```
flexChart.ChartType = C1.Chart.ChartType.LineSymbols;
```

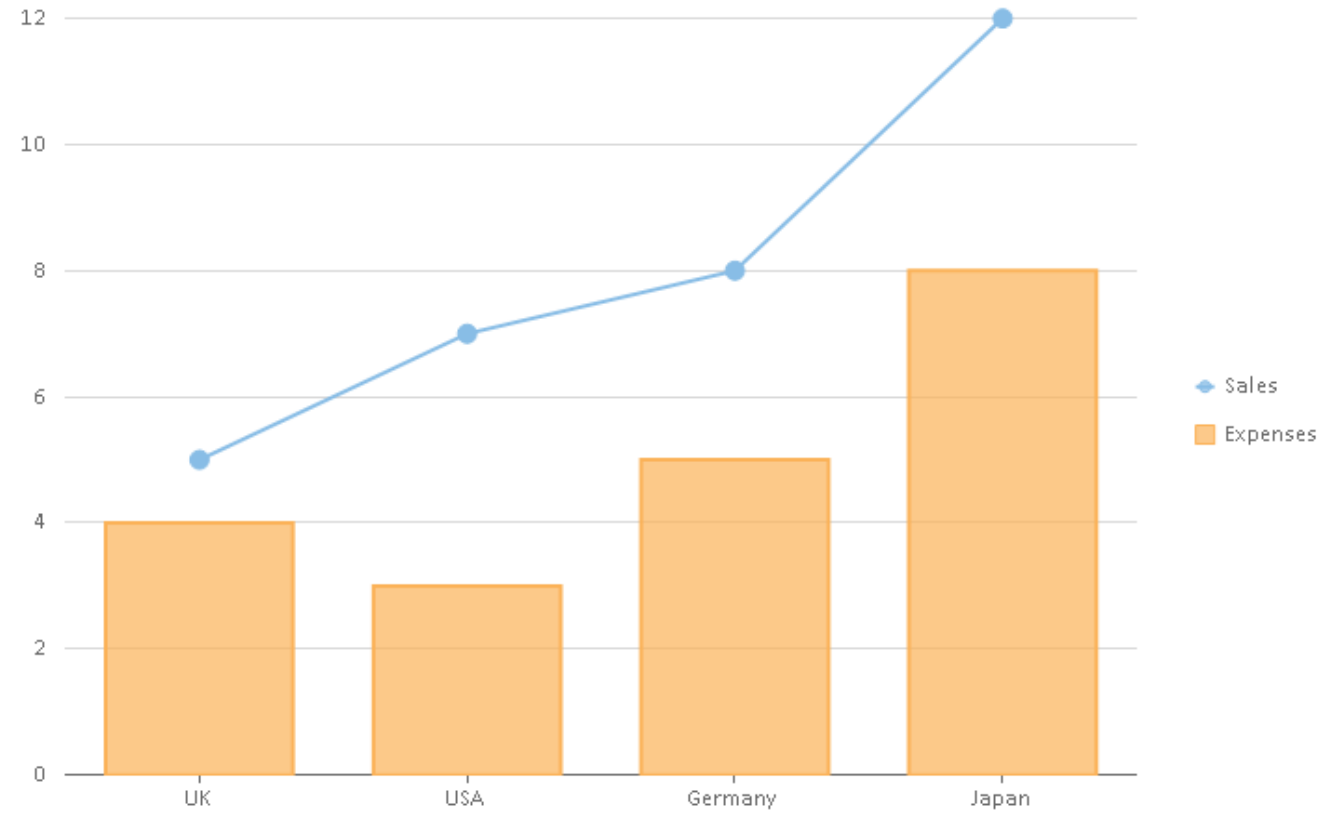
Mixed

FlexChart allows you to create mixed charts that offer two key advantages, as follows:

- **Combining chart types:** Combine two or more chart types in a single chart, for instance, area-bar, bar-line, bar-scatter etc. Plot different metrics in a chart using different chart types and let the end user interpret data easily. In FlexChart, specify a chart type for each series to combine several chart types. To specify the chart type for a series, set the ChartType property of the Series class. Setting this property overrides the ChartType property set for the chart.
- **Plotting multiple datasets:** Plot data from multiple datasets in a single chart by specifying data source for a series. This is useful when the data to plot lies at multiple places. To specify the data source for a series, set the ItemsSource property of the Series class. Setting this property overrides the ItemsSource property set for the chart.

Copyright © 2019 GrapeCity, Inc. All rights reserved.

The following image displays a mixed chart that combines column and line symbols chart types. The chart plots and compares sales and expenses data of four countries.



The following code sets the Column chart type for FlexChart and overrides it by setting the LineSymbols chart type for the Sales series, thereby implementing mixed charts.

- **XAML**

```
<c1:C1FlexChart x:Name="flexChart"
                BindingX="Country"
                ChartType="Column"
                ItemsSource="{Binding DataContext.Data}">
    <c1:Series SeriesName="Sales"
               Binding="Sales"
               ChartType="LineSymbols"/>
    <c1:Series SeriesName="Expenses"
               Binding="Expenses"/>
</c1:C1FlexChart>
```

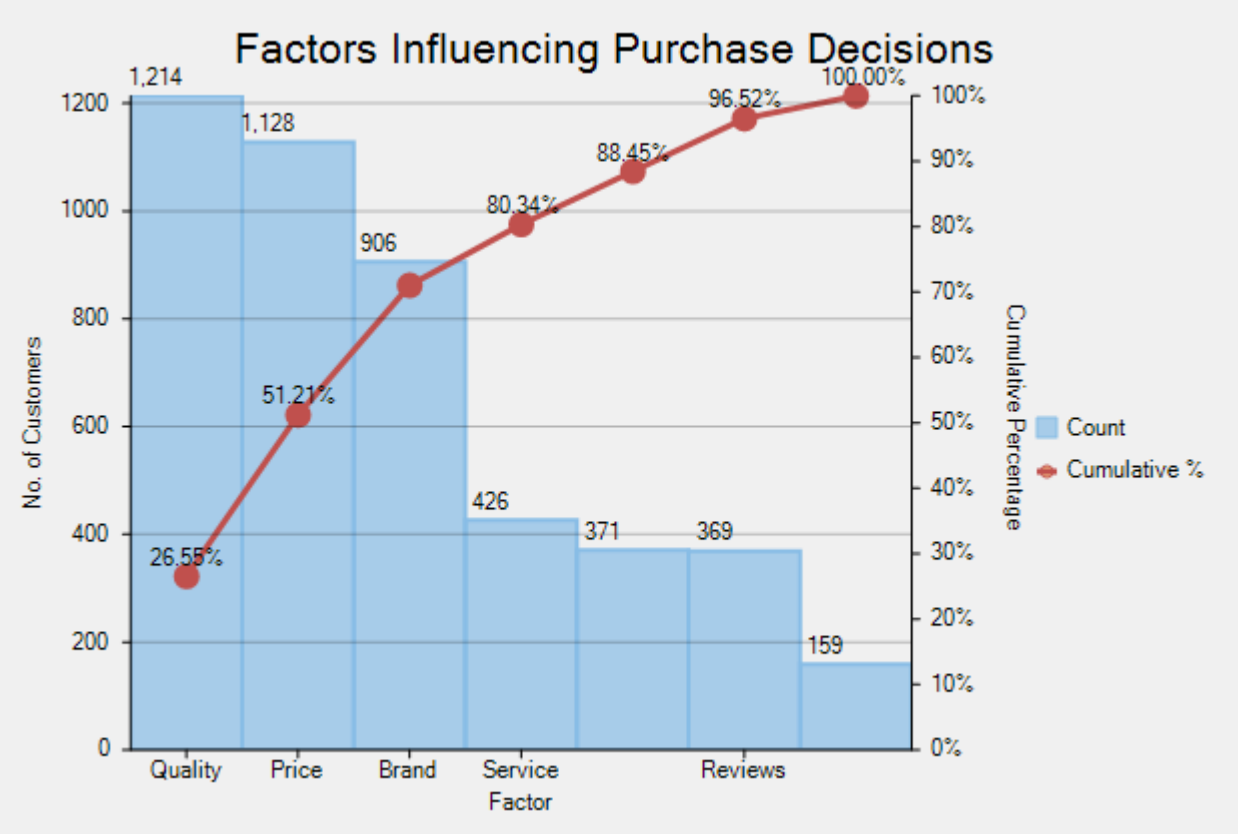
Pareto Chart

Pareto chart is a type of chart that contains both bar and a line chart. It is a vertical bar chart in which values are plotted in decreasing order of relative frequency from left to right. The categories or factors that represent the bigger bars on the left are more important than those on the right. The line chart plots the cumulative total percentage of frequencies that are represented by the bars.

Pareto chart is essentially used in scenarios where the data is broken into different categories, and when the developer needs to highlight the most important factors from a given set of factors. For example, quality control, inventory control, and customer grievance handling are some areas where Pareto chart analysis can be frequently used.

Copyright © 2019 GrapeCity, Inc. All rights reserved.

In FlexChart, Pareto chart can be easily created by combining RangedHistogram chart with any of Line, Spline, LineSymbol, or SplineSymbol chart. First, plot the relative frequency on a RangedHistogram in descending order. Then, calculate the cumulative relative frequency in percentage using original data to create another series which is plotted on any of the Line, Spline, LineSymbol, or SplineSymbol chart. This forms Pareto line of the chart which helps in identifying the added contribution of each category.



To implement Pareto chart using the FlexChart control, see **FlexChartExplorer** sample. The samples are available in the default installation folder - *Documents\ComponentOne Samples*

RangedHistogram

RangedHistogram is a modern Excel-like histogram chart that helps visualize frequency distribution on y axis, against **ranged** x axis. Like Histogram chart type, bins are created by dividing the raw data values into a series of consecutive, non-overlapping intervals. Based on the number of values falling in a particular bin, frequencies are then plotted as rectangular columns against x-axis.

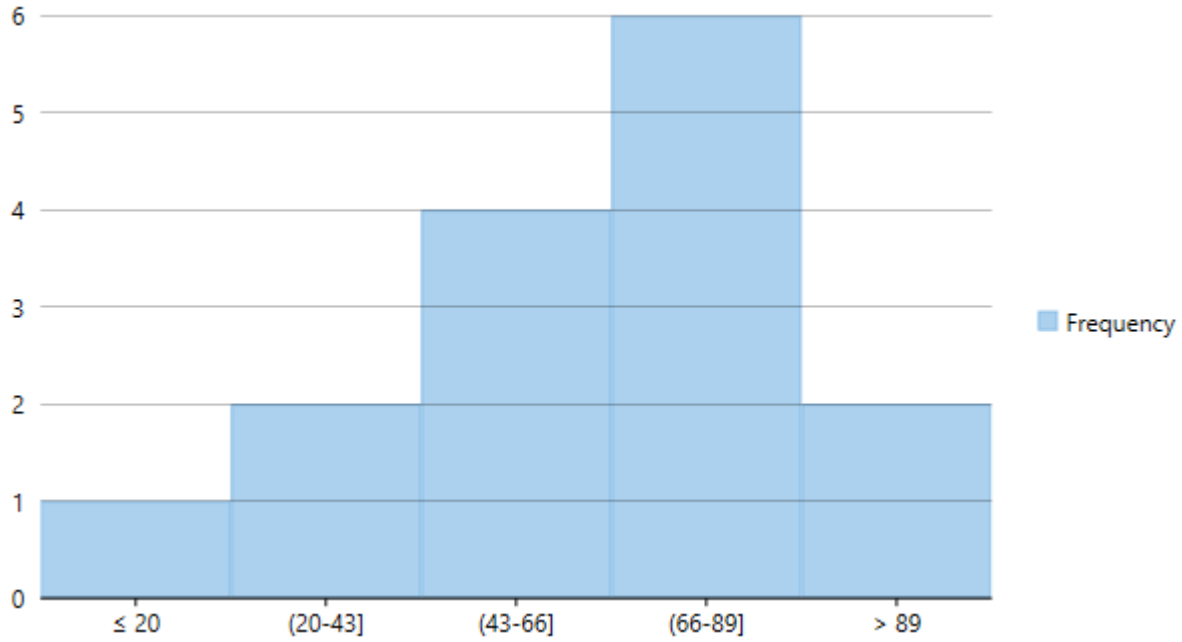
RangedHistogram plots frequency distribution for the provided data in non-category and category modes.

Non-Category Mode

In non-category mode, the original data points are binned into intervals or ranges. These intervals are then plotted on x-axis, and y-axis shows frequency distribution for respective ranges. FlexChart automatically calculates the intervals in which your data is grouped.
However, you can control this behavior by specifying the HistogramBinning through BinMode property. Moreover, you can further set BinWidth, NumberOfBins, values for UnderflowBin and OverflowBin, and specify whether to ShowUnderflowBin and ShowOverflowBin.

The following image illustrates frequency distribution for units sold of various products of a retail store in non-category mode.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



To create a RangedHistogram, for a given data in non-category mode, you need to add the RangedHistogram series and set the ChartType property to **RangedHistogram**, as shown in the following code snippet.

- **Xaml**

```
<Chart:C1FlexChart x:Name="flexChart"
                       ChartType="RangedHistogram"
                   ItemsSource="{Binding DataContext.Data}"
                   Binding="Value">
    <Chart:RangedHistogram x:Name="RangedhistogramSeries"
                           SeriesName="Frequency"
                           BinMode="NumberOfBins"
                           NumberOfBins="5"
                           OverflowBin="89"
                           UnderflowBin="20"
                           ShowOverflowBin="True"
                           ShowUnderflowBin="True"/>
</Chart:C1FlexChart>
```

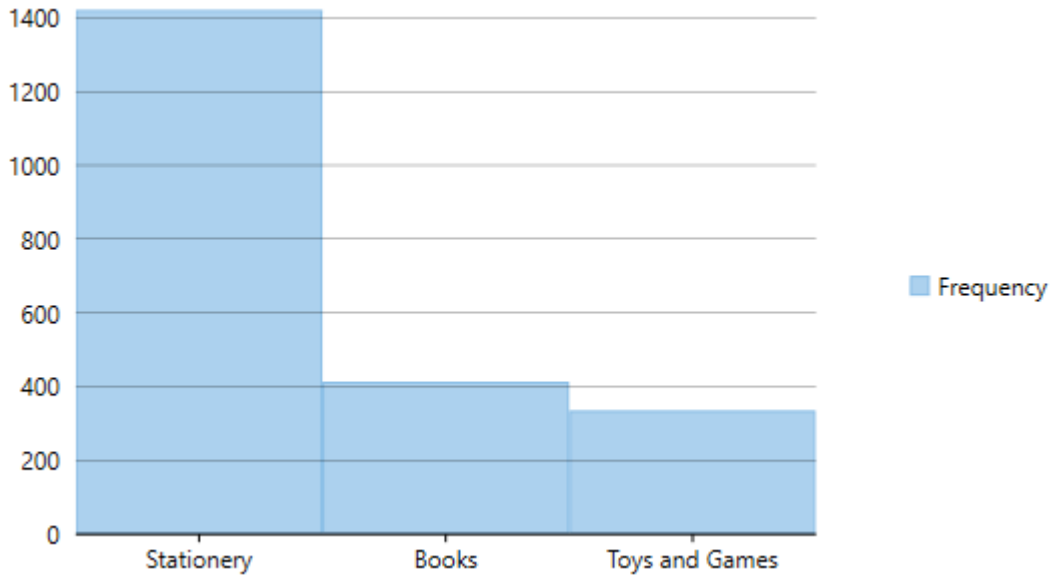
Category Mode

In Category mode, frequency data is exclusively grouped in categories (which are plotted on x-axis) as provided by the original data and y-axis depicts cumulative frequency for the respective categories. Category mode is enabled for RangedHistogram series by setting the BindingX property.

In this mode, BinMode, BinWidth, NumberOfBins, OverflowBin, and UnderflowBin properties for RangedHistogram series are ignored.

The following image illustrates frequency distribution for units sold of 3 categories of products Stationery items, Books, and Toys and Games of a retail store in category mode.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



To create a RangedHistogram for a given data in category mode, you need to add the RangedHistogram series, set the ChartType property to **RangedHistogram** and set the BindingX property, as shown in the following code snippet.

- **Xaml**

```
<Chart:C1FlexChart x:Name="flexChart"
                   ChartType="RangedHistogram"
                   ItemsSource="{Binding DataContext.Data}"
                   Binding="Value"
                   BindingX="Name">
  <Chart:RangedHistogram x:Name="RangedhistogramSeries"
                         SeriesName="Frequency" />
</Chart:C1FlexChart>
```

Note that unlike traditional Histogram, other chart types cannot be plotted using the same x axis values as RangedHistogram.

Scatter

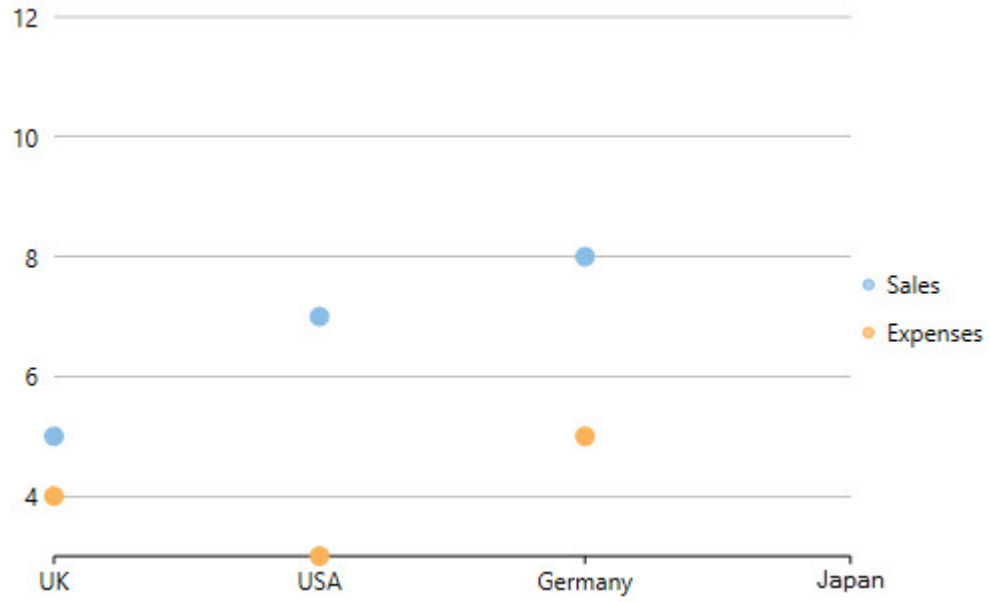
The Scatter Chart, which is also known as the XY Chart, depicts relationship among items of different data series. In simple terms, it is a plot of X values and Y values along the two axes. The data points are not connected and can be customized using different symbols. This chart type is normally used to represent scientific data, and can highlight the deviation of assembled data from predicted data or result.

To create the Scatter Chart, you can set the ChartType property to Scatter at design-time, in the code behind, or in XAML.

Set the Stacking property to Stacked or Stacked100pc to create the stacking Scatter Chart.

Scatter Chart

Copyright © 2019 GrapeCity, Inc. All rights reserved.



Below is the code implementing the scenario:

XAML

```
<c1:C1FlexChart x:Name="flexChart"
                BindingX="Country"
                ItemsSource="{Binding DataContext.Data}"
                ChartType="Scatter">
    <c1:C1FlexChart.Series>
        <c1:Series SeriesName="Sales"
                   Binding="Sales"/>
        <c1:Series SeriesName="Expenses"
                   Binding="Expenses"/>
    </c1:C1FlexChart.Series>
</c1:C1FlexChart>
```

Code

C# copyCode

```
flexChart.ChartType = C1.Chart.ChartType.Scatter;
```

Spline

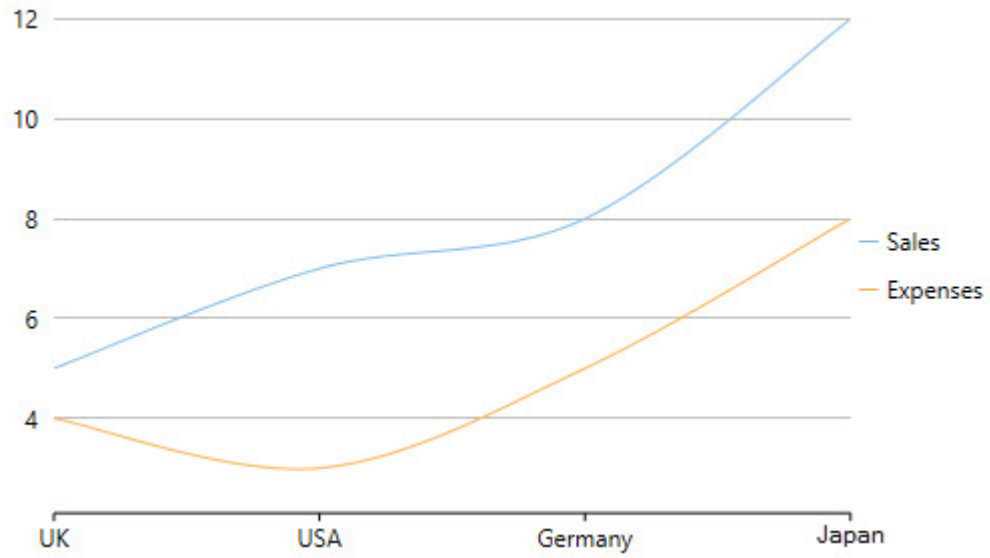
The Spline Chart is similar to the line chart except that it connects data points by using splines rather than straight lines. The chart is used as an alternative to the line chart, but more specifically for representing data that requires the use of curve fittings.

You need to set the ChartType property to Spline either from the Properties window, in the code behind, or in XAML to create the Spline Chart.

You can set the Stacking property to Stacked or Stacked100pc to create the stacking Spline Chart.

Spline Chart

Copyright © 2019 GrapeCity, Inc. All rights reserved.



Here is the code demonstrating the implementation:

XAML

```
<c1:C1FlexChart x:Name="flexChart"
                BindingX="Country"
                ItemsSource="{Binding DataContext.Data}"
                ChartType="Spline">
    <c1:C1FlexChart.Series>
        <c1:Series SeriesName="Sales"
                   Binding="Sales"/>
        <c1:Series SeriesName="Expenses"
                   Binding="Expenses"/>
    </c1:C1FlexChart.Series>
</c1:C1FlexChart>
```

Code

C# copyCode

```
flexChart.ChartType = C1.Chart.ChartType.Spline;
```

SplineArea

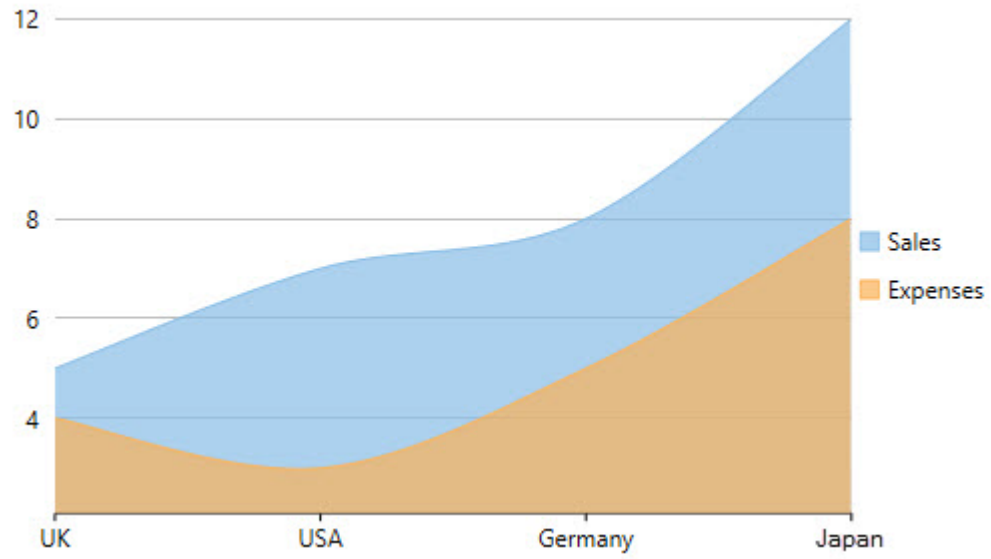
The SplineArea chart is just like the area chart with the only difference in the manner in which data points are connected. The SplineArea chart connects data points by using splines instead of straight lines, and fills the area enclosed by the splines.

Set the ChartType property to SplineArea in the Properties window, in the code behind, or in XAML to create the SplineArea Chart.

To create the stacking SplineArea Chart, set the Stacking property to Stacked or Stacked100pc.

SplineArea Chart

Copyright © 2019 GrapeCity, Inc. All rights reserved.



Here is the implementation in code:

XAML

```
<c1:C1FlexChart x:Name="flexChart"
                BindingX="Country"
                ItemsSource="{Binding DataContext.Data}"
                ChartType="SplineArea">
    <c1:C1FlexChart.Series>
        <c1:Series SeriesName="Sales"
                   Binding="Sales"/>
        <c1:Series SeriesName="Expenses"
                   Binding="Expenses"/>
    </c1:C1FlexChart.Series>
</c1:C1FlexChart>
```

Code

C# copyCode

```
flexChart.ChartType = C1.Chart.ChartType.SplineArea;
```

SplineSymbols

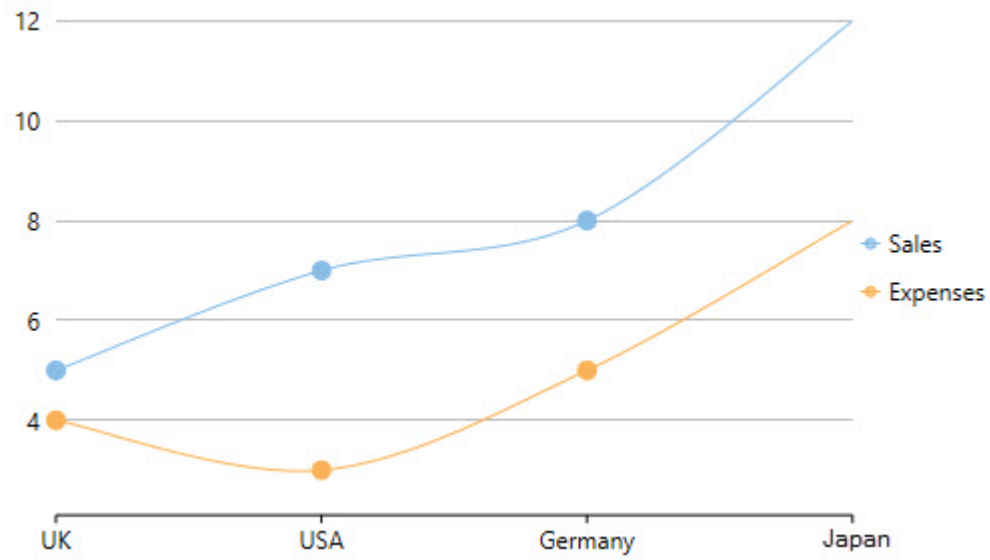
The SplineSymbols Chart combines the Spline Chart and the Scatter Chart. The chart plots data points by using symbols and connects those data points by using splines.

To create the SplineSymbols Chart, you can set the ChartType property to SplineSymbols in the Properties window, in XAML, or programmatically.

Set the Stacking property to Stacked or Stacked100pc to create the stacking SplineSymbols Chart.

SplineSymbols Chart

Copyright © 2019 GrapeCity, Inc. All rights reserved.



See the following code for implementation:

XAML

```
<c1:C1FlexChart x:Name="flexChart"
                BindingX="Country"
                ItemsSource="{Binding DataContext.Data}"
                ChartType="SplineSymbols">
    <c1:C1FlexChart.Series>
        <c1:Series SeriesName="Sales"
                   Binding="Sales"/>
        <c1:Series SeriesName="Expenses"
                   Binding="Expenses"/>
    </c1:C1FlexChart.Series>
</c1:C1FlexChart>
```

Code

C# copyCode

```
flexChart.ChartType = C1.Chart.ChartType.SplineSymbols;
```

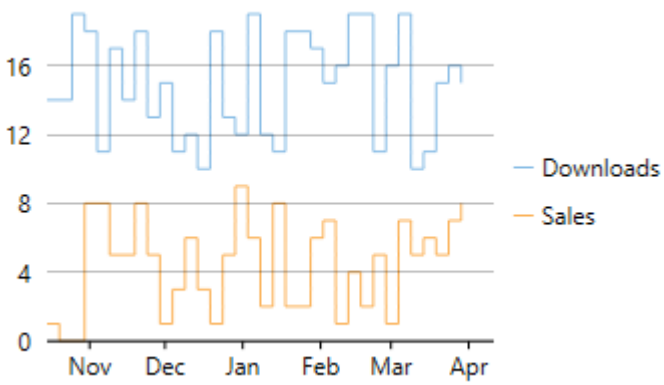
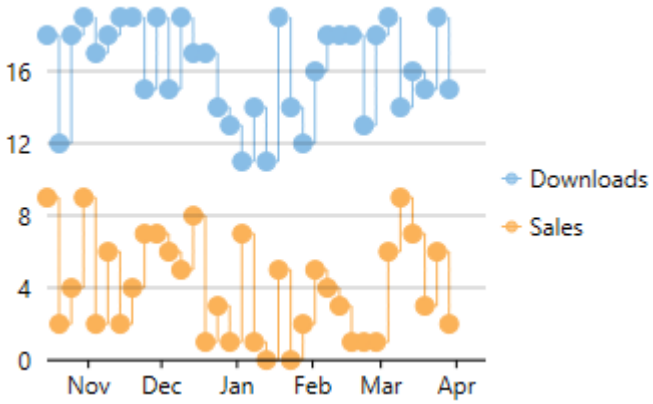
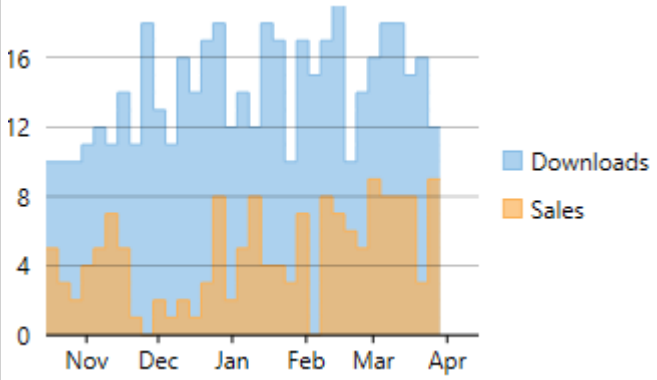
Step

Step charts use horizontal and vertical lines to present data that show sudden changes along y-axis by discrete amount. These charts help display changes that are sudden and irregular but stay constant till the next change. Step charts enable judging trends in data along with the duration for which the trend remained constant.

Consider a use case where you want to visualize and compare weekly sales and units downloaded of a software. As both of these values vary with discrete amounts, you can use step chart to visualize them. As shown in the image below, apart from depicting the change in sales these charts also show the exact time of change and the duration for which sales were constant. Moreover, you can easily identify the magnitude of respective changes by simply looking at the chart.

FlexChart supports Step chart, StepSymbols chart, and StepArea or filled step chart. The following table gives detailed explanation of these chart types.

Copyright © 2019 GrapeCity, Inc. All rights reserved.

 **Step Chart**	Step chart is similar to the Line chart, except that Line chart uses shortest distance to connect consecutive data points, while Step chart connects them with horizontal and vertical lines. These horizontal and vertical lines give the chart step-like appearance. While the line charts depict change and its trend, the Step charts also help in judging the magnitude and the intermittent pattern of the change.
 **StepSymbols Chart**	StepSymbols chart combines the Step chart and the Scatter chart. FlexChart plots data points by using symbols and connects those data points with horizontal and vertical step lines. Here, the data points are marked using symbols and, therefore, help mark the beginning of an intermittent change.
 **StepArea Chart**	StepArea chart combines the Step chart and the Area chart. It is similar to Area chart with the difference in the manner in which data points are connected. FlexChart plots the data points using horizontal and vertical step lines, and then fills the area between x-axis and the step lines. These are based on Step charts, and are commonly used to compare discrete and intermittent changes between two or more quantities. This gives the chart stacked appearance, where related data points of the multiple series seem stacked above the other. For example, number of units downloaded and sales of a software for a particular time duration can be easily compared as shown in the image.

To create Step chart, you need to populate appropriate data in chart and set the ChartType property to **Step** in XAML, at design-time, or using code. However, to create a StepArea or a StepSymbol chart, you need to set the **ChartType** property to **StepArea** or **StepSymbols** respectively.

- **XAML**

```
<c1:C1FlexChart x:Name="flexChart"
                BindingX="Date"
                ItemsSource="{Binding DataContext.Data}"
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
                ChartType="Step"
                HorizontalAlignment="Stretch" Margin="5,5,5,195">
    <c1:C1FlexChart.Series>
        <c1:Series SeriesName="Downloads" Binding="Downloads"></c1:Series>
        <c1:Series SeriesName="Sales" Binding="Sales"></c1:Series>
    </c1:C1FlexChart.Series>
</c1:C1FlexChart>
```

C#

```
flexChart.ChartType = C1.Chart.ChartType.Step;
```

Working with FlexChart

To work with **FlexChart for WPF** and use it for developing your applications, you should know how to leverage several features and functionality offered by the control.

This section provides important conceptual, task-based information on features and functionality offered by FlexChart.

The below-mentioned links take you to the sections that discuss the different ways in which you can work with FlexChart.

- Data
- Appearance
- End-User Interaction
- FlexChart Elements
- Trend Lines
- Export

Data

Data is the first and foremost requirement of a chart. Without data, a chart cannot possibly visualize or display anything. Thus, while working with a chart, your first job is to get your chart to display data, so that you can work with and interpret the data accordingly.

When it comes to your chart data, there are two primary stages that sum up the specification, representation, and interpretation of the data:

- Providing data
- Plotting data

Access these sections to go through these stages with reference to FlexChart at length.

Providing Data

To get your chart to plot data, you first need to provide data to the chart.

The most common and widely used approach for providing data to the chart is data binding.

Click the following link to know how you can bind data to FlexChart:

- Binding data using a data source

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Binding Data Using a Data Source

Binding data means connecting one or more data consumers to a data provider in a synchronized manner. When data bound, the chart uses all of the bound data as its source of data for the specified series, and represents the data on the chart surface as per the series and chart properties.

Since there is a layer between the data source and the actual chart, the data often needs to be summarized before it can be plotted; however, the data to be plotted sometimes may already be available in a data view or another data source object. And therefore, you can bind the chart directly to the data source object in such cases.

To bind the **FlexChart** control to the data source, you first need to set the ItemsSource property to the data source object. Next, you need to bind individual series of the chart to the fields present in the data source object by using the BindingX and the Binding property.

> To specify the binding source, you need to add the **DataContext = "{Binding RelativeSource= {RelativeSource Mode=Self}}"** markup in the **<Window>** tag of the **MainWindow.xaml** file.

The below-mentioned code uses a class DataCreator to generate the data for the chart.

Here is the code that demonstrates a fully-functional program implementing data binding:

XAML

```
<Window
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:local="clr-namespace:Data_Binding"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:Chart="using:C1.Xaml.Chart"
        xmlns:Xaml="using:C1.Xaml"
        xmlns:c1="http://schemas.componentone.com/winfx/2006/xaml"
        xmlns:local1="clr-namespace:Data_Binding"
        x:Name="window"
        x:Class="Data_Binding.MainWindow"
        DataContext="{Binding RelativeSource={RelativeSource Mode=Self}}"
        mc:Ignorable="d">

    <Grid>

        <c1:C1FlexChart x:Name="flexChart"
                        BindingX="Fruit"
                        ItemsSource="{Binding DataContext.Data}"
                        ChartType="Bar">
            <c1:C1FlexChart.Series>
                <c1:Series SeriesName="Function1"
                           Binding="March"/>
                <c1:Series SeriesName="Function2"
                           Binding="April"/>
            </c1:C1FlexChart.Series>
        </c1:C1FlexChart>

    </Grid>
```

Code

DataCreator.cs copyCode

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
class DataCreator
{
    public static List<FruitDataItem> CreateFruit()
    {
        var fruits = new string[] { "Oranges", "Apples", "Pears", "Bananas" };
        var count = fruits.Length;
        var result = new List<FruitDataItem>();
        var rnd = new Random();
        for (var i = 0; i < count; i++)
            result.Add(new FruitDataItem()
            {
                Fruit = fruits[i],
                March = rnd.Next(20),
                April = rnd.Next(20),
                May = rnd.Next(20),
            });
        return result;
    }

}
public class FruitDataItem
{
    public string Fruit { get; set; }
    public double March { get; set; }
    public double April { get; set; }
    public double May { get; set; }
}

public class DataPoint
{
    public double XVals { get; set; }
    public double YVals { get; set; }
}
```

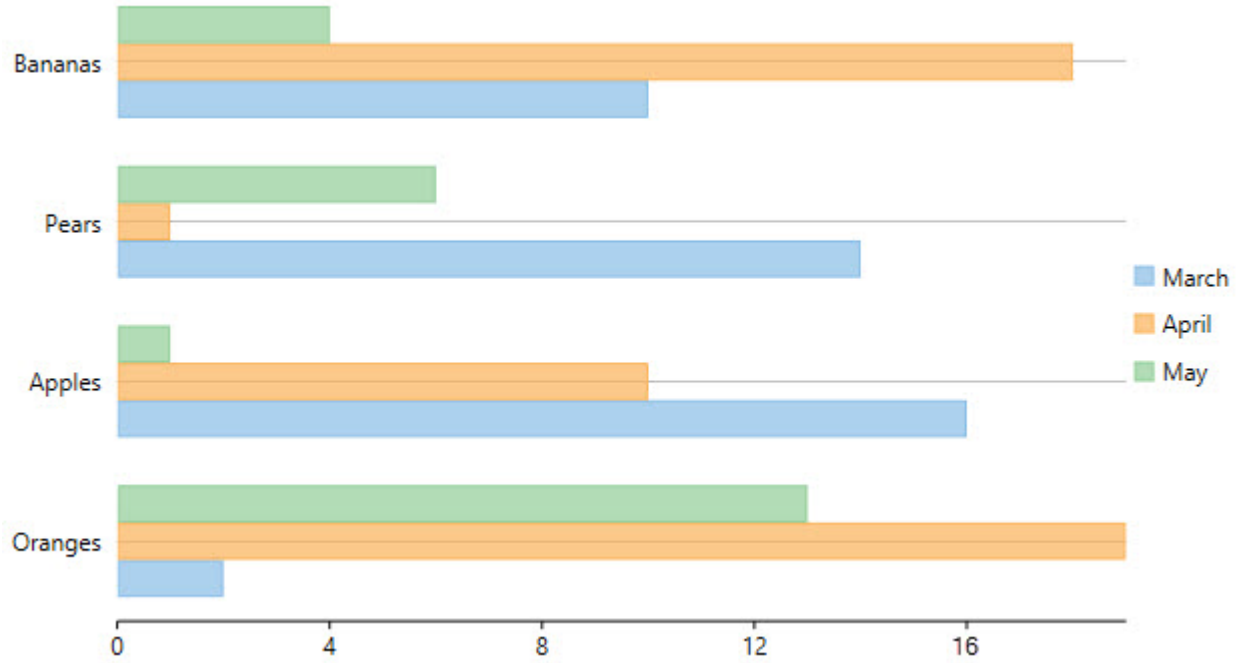
MainWindow.xaml.cs copyCode

```
public partial class MainWindow : Window
{

    #region plotdata
    private List<FruitDataItem> _fruits;

    public MainWindow()
    {
        this.InitializeComponent();

    }

    public List<FruitDataItem> Data
    {
        get
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
        {
            if (_fruits == null)
            {
                _fruits = DataCreator.CreateFruit();
            }

            return _fruits;
        }
    }
```

The following output appears once you have run the code:



Plotting Data

FlexChart plots data bound in the form of fields or data arrays when relevant values are set in the BindingX and the Binding property.

You require setting the values in the BindingX and the Binding property as per the desired chart type . For instance, in case of the Scatter Chart, you need to set a single value (field) in both the BindingX and the Binding property. However, in case of the Bubble Chart, you need to set a single value (field) in the BindingX property and two values (fields, one for specifying Y-values and another for specifying the size of the bubble) in the Binding property.

See the following code snippets for reference:

1. In case of Scatter Chart

- **XAML**

```
<c1:C1FlexChart x:Name="flexChart"
                BindingX="Country"
                ItemsSource="{Binding DataContext.Data}"
                ChartType="Scatter">
    <c1:C1FlexChart.Series>
        <c1:Series SeriesName="Sales"
                   Binding="Sales"/>
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
        <c1:Series SeriesName="Expenses"
                   Binding="Expenses"/>
    </c1:C1FlexChart.Series>
</c1:C1FlexChart>
```

2. In case of Bubble Chart

- **XAML**

```
<c1:C1FlexChart x:Name="flexChart"
                BindingX="X"
                ItemsSource="{Binding DataContext.Data}"
                ChartType="Bubble">
    <c1:C1FlexChart.Series>
        <c1:Series SeriesName="Bubble"
                   Binding="Y,Size"/>
    </c1:C1FlexChart.Series>
</c1:C1FlexChart>
```

Once the data is plotted, you can work on it to visualize data that suits your requirements.

Go through the sections given below to learn how to customize series and plot irregular data.

- Customizing Series
- Interpolating Null Values

Customizing Series

Once the series have been displayed in the chart, you can customize the displayed series to manage the same more efficiently.

FlexChart allows you to customize series by showing or hiding a series either in the Plot Area or the Legend or both.

The following sections detail the different customizations:

- Showing or Hiding a Series

Showing or Hiding a Series

If there are hundreds of series to be displayed in your chart, you would certainly need to manage the same due to the space limitation of the chart.

In **FlexChart**, you can manage series by using the Visibility property of a series. The **Visibility** property accepts values of the SeriesVisibility enumerated type.

You can set the property to the following different values to show or hide a series:

Value	Description
SeriesVisibility.Visible	The series is displayed in the Plot as well as the Legend.
SeriesVisibility.Plot	The series is displayed in the Plot, but hidden in the Legend.
SeriesVisibility.Legend	The series is displayed in the Legend, but hidden in the Plot.

Copyright © 2019 GrapeCity, Inc. All rights reserved.

SeriesVisibility.Hidden	The series is hidden in the Plot as well as the Legend.

Here's the code snippet showing how to set the Visibility property:

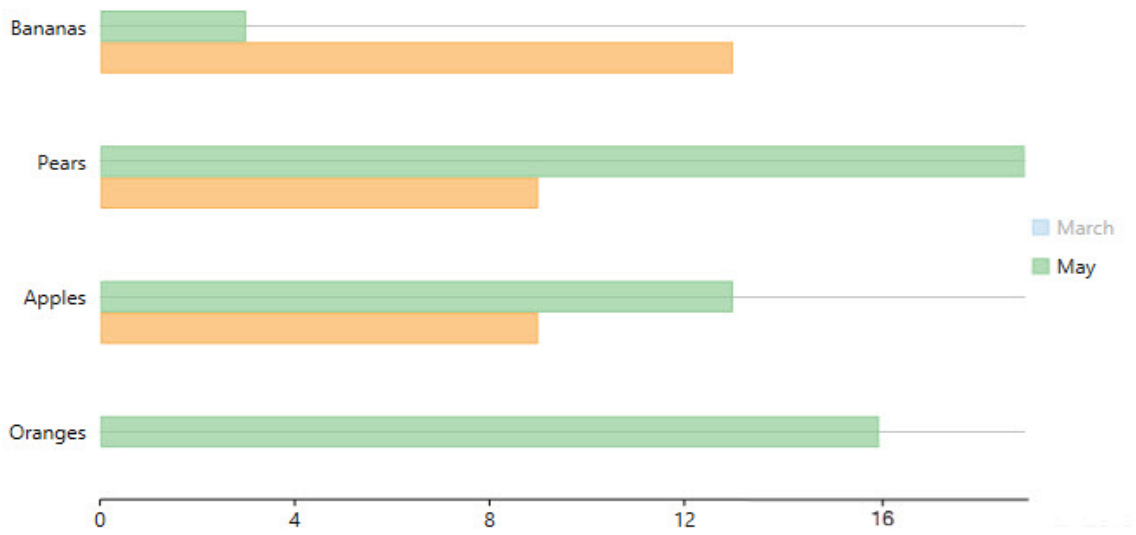
XAML

```
<c1:Series SeriesName="March"
           Binding="March"
           Visibility="Legend"/>
<c1:Series SeriesName="April"
           Binding="April"
           Visibility="Plot"/>
<c1:Series SeriesName="May"
           Binding="May"/>
```

Code

C# copyCode

```
series1.SeriesName = "March";
series2.SeriesName = "April";

series1.Visibility = C1.Chart.SeriesVisibility.Legend;
series2.Visibility = C1.Chart.SeriesVisibility.Plot;
```



Interpolating Null Values

Often, there are null values in the data fields of a data table that you bind to **FlexChart** for plotting data. Due to the presence of null values, FlexChart creates gaps once it has plotted the data. The gaps present in the plotted data make data look inconsistent and incomplete.

FlexChart allows you to deal with such inconsistencies by using the InterpolateNulls property. You can

Copyright © 2019 GrapeCity, Inc. All rights reserved.

set the InterpolateNulls property, so that the chart automatically fills in gaps created by null values in data.

> The **InterpolateNulls** property is applicable only for the Line Chart and the Area Chart.

Here is how you can set the InterpolateNulls property:

- **C#**

```
flexChart.Options.InterpolateNulls = true;
```

Appearance

The appearance of a chart determines its overall look and feel. A good and clean appearance draws your audiences toward the visual representation of your data. And it also adds to the ease of interpreting the data.

You can customize the appearance of FlexChart in a variety of ways that are mentioned below:

- Animation
- Colors
- Fonts
- Symbol Styles for Series

Colors

Colors are used to enhance the visual impact of a chart. You can customize colors by choosing colors interactively, setting chart palette, specifying RGB values, specifying hue, saturation, and brightness, or using transparent colors.

FlexChart lets you customize colors for the entire chart as well as the following elements:

- Series
- Header and Footer
- Legend
- Plot Area
- Label

Click the links given below to learn how to use colors in different ways.

- Choosing Colors Interactively
- Setting Chart Palette
- Specifying RGB Colors
- Specifying Hue, Saturation, and Brightness
- Using Transparent Colors

Copyright © 2019 GrapeCity, Inc. All rights reserved.

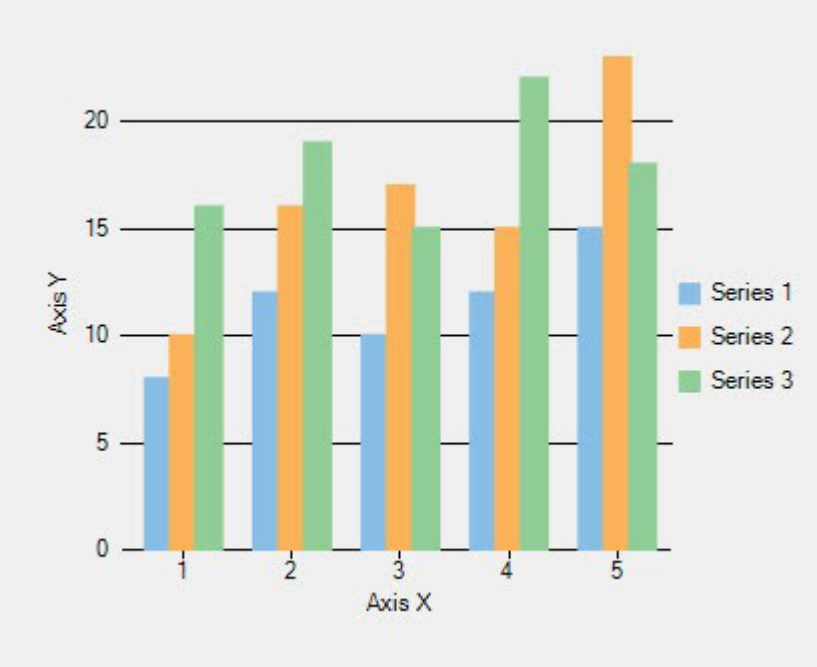
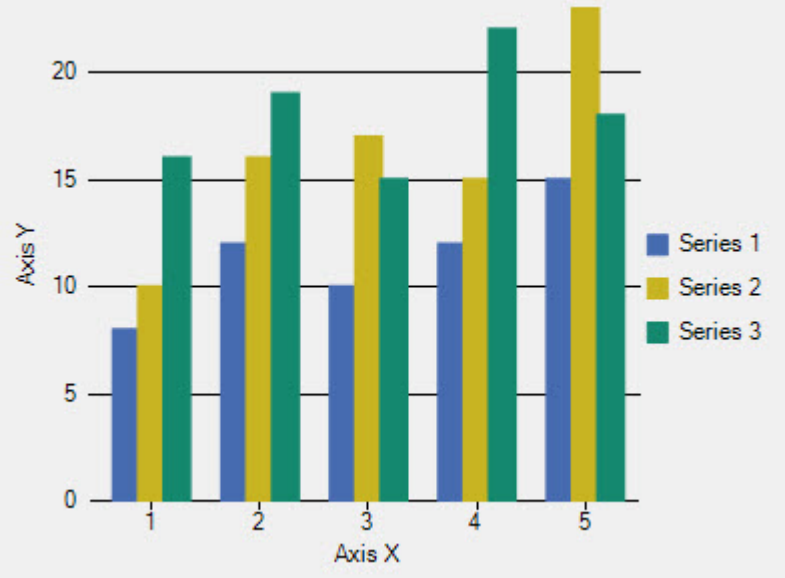
Choosing Colors Interactively

Colors can be chosen interactively by using .NET's color dialog that works like the standard windows color dialog. You can choose from Windows basic colors or customer colors, or you can interactively choose from a full color spectrum.

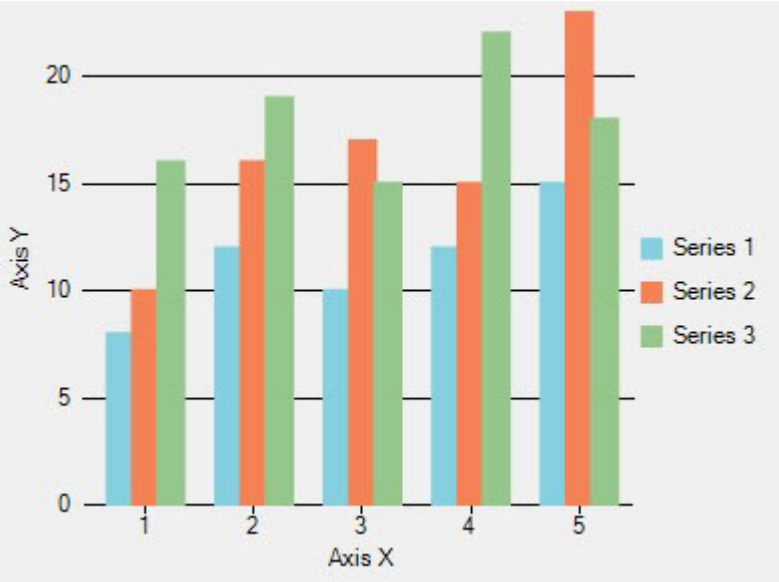
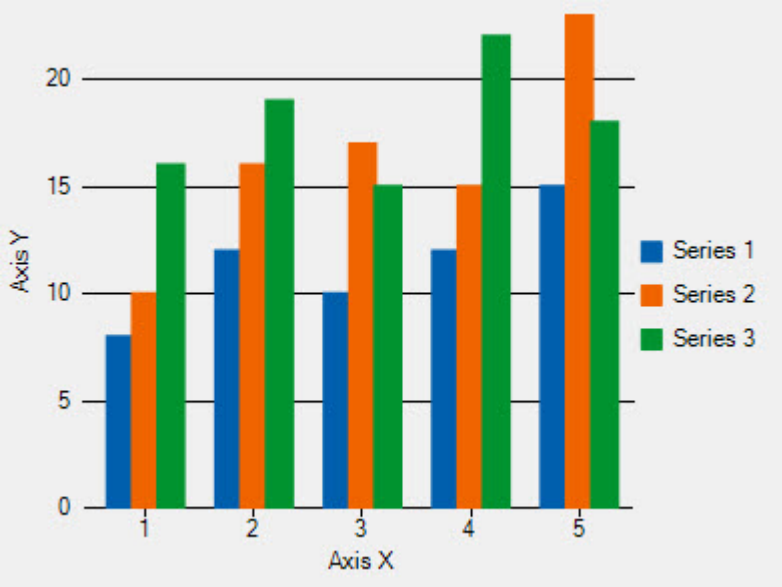
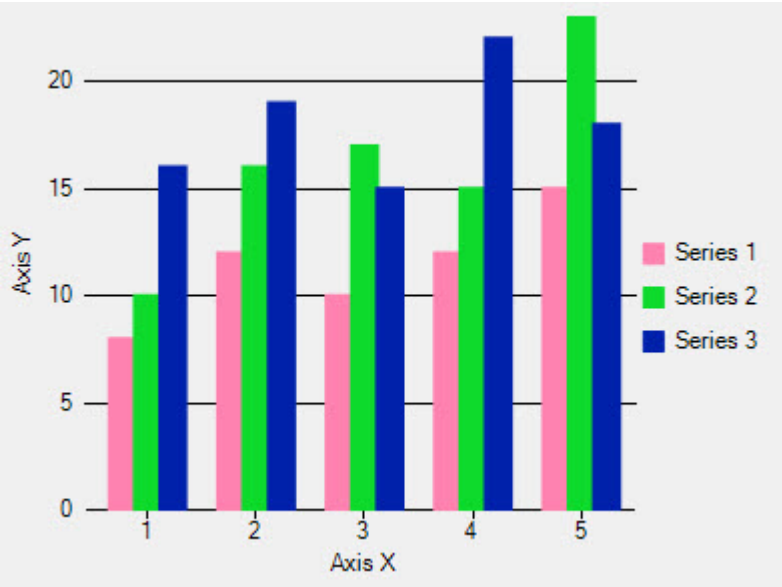
Setting FlexChart Palette

You can set the desired **FlexChart** palette by using the Palette property. By default, FlexChart uses the **Palette.Standard** setting that specifies the standard chart palette.

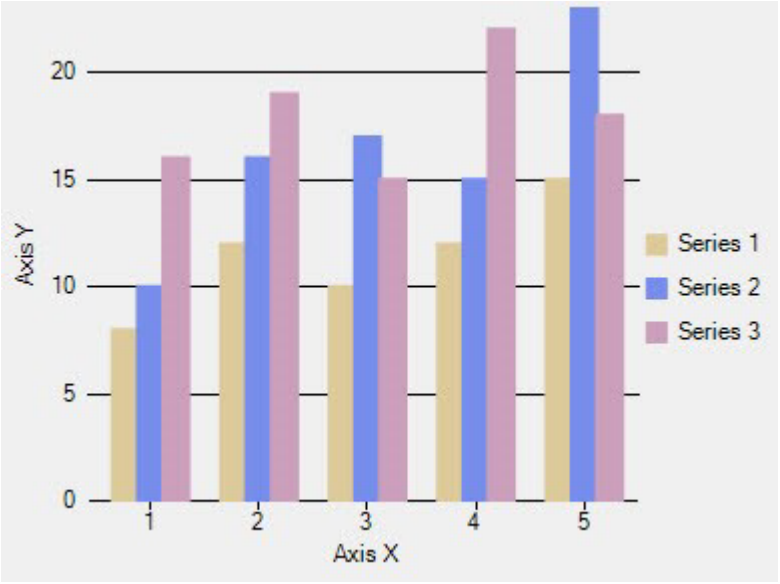
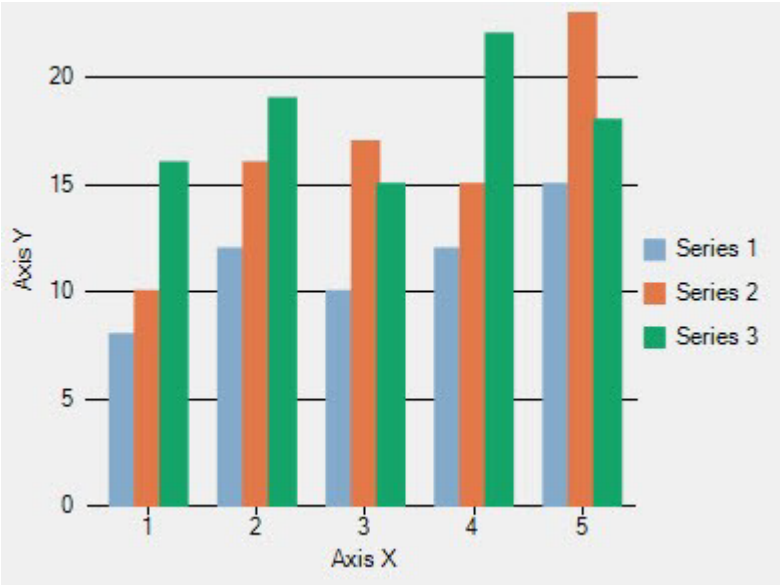
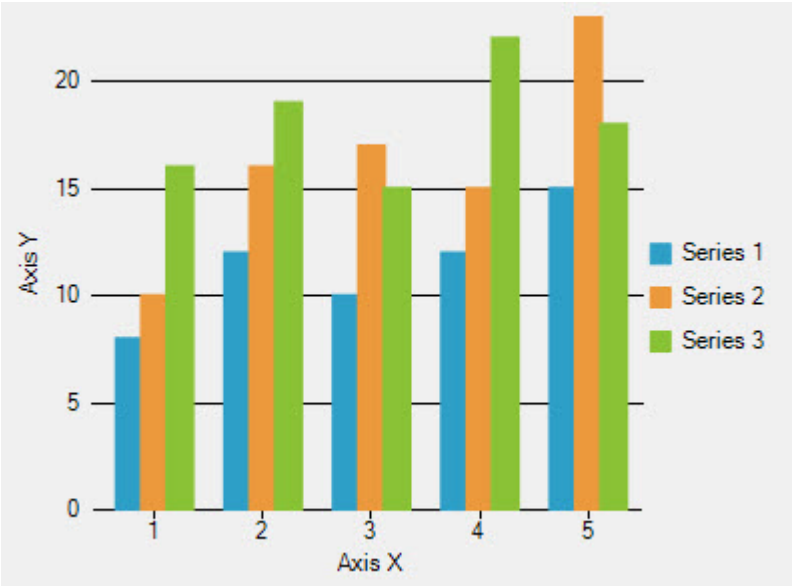
Here are the available palettes in FlexChart:

Palette Setting	Preview
Standard (default)	
Cocoa	

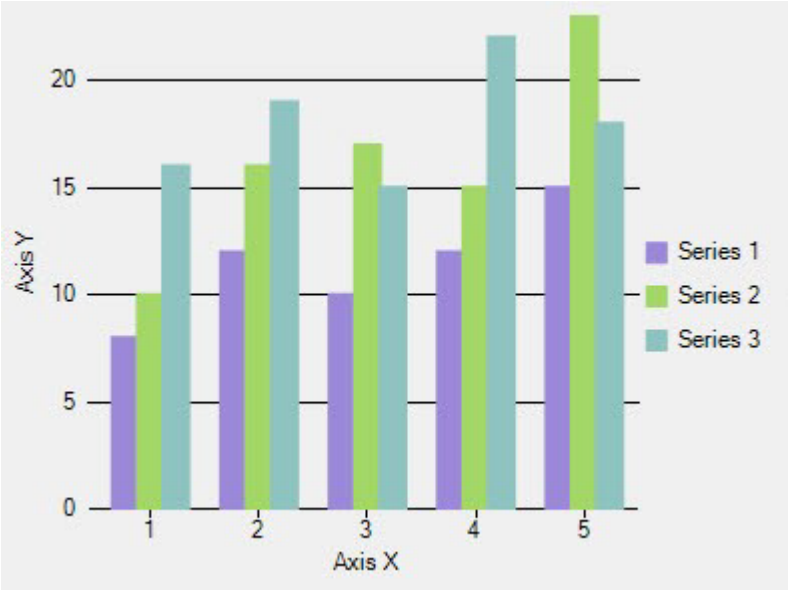
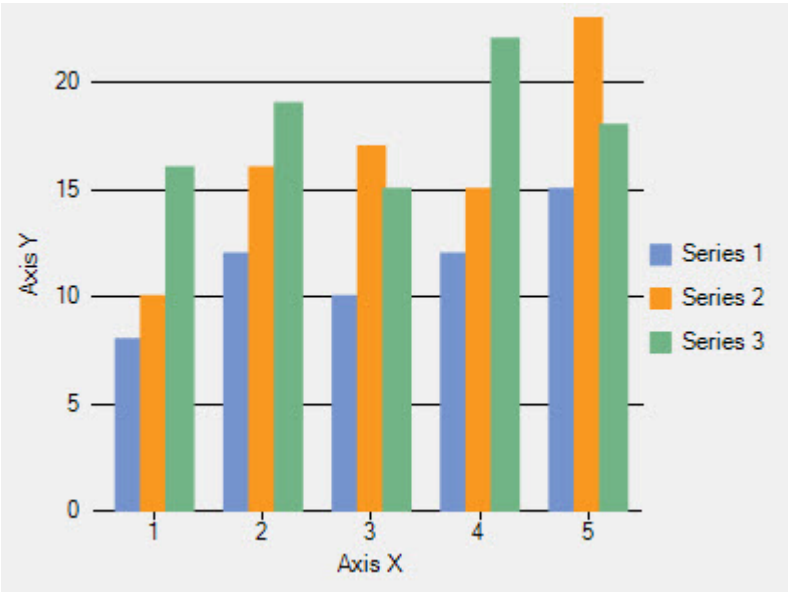
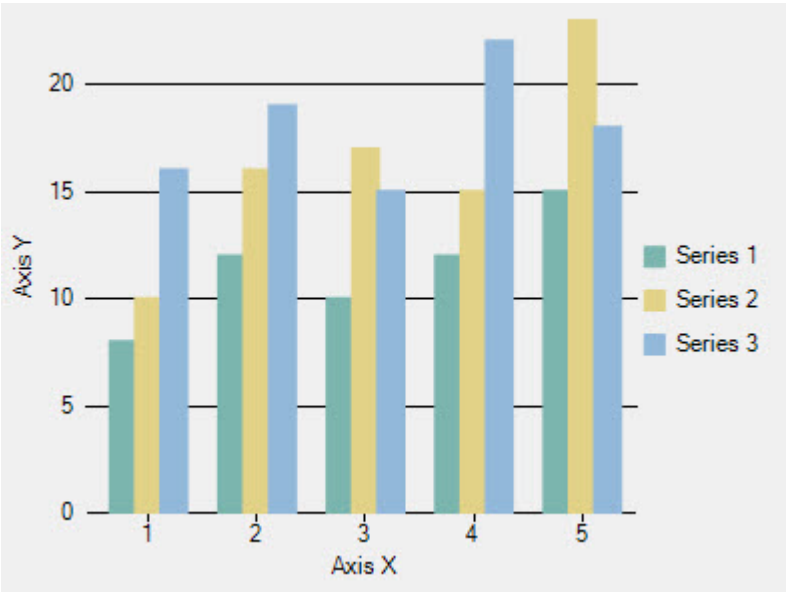
Copyright © 2019 GrapeCity, Inc. All rights reserved.

Coral	
Dark	
Highcontrast	

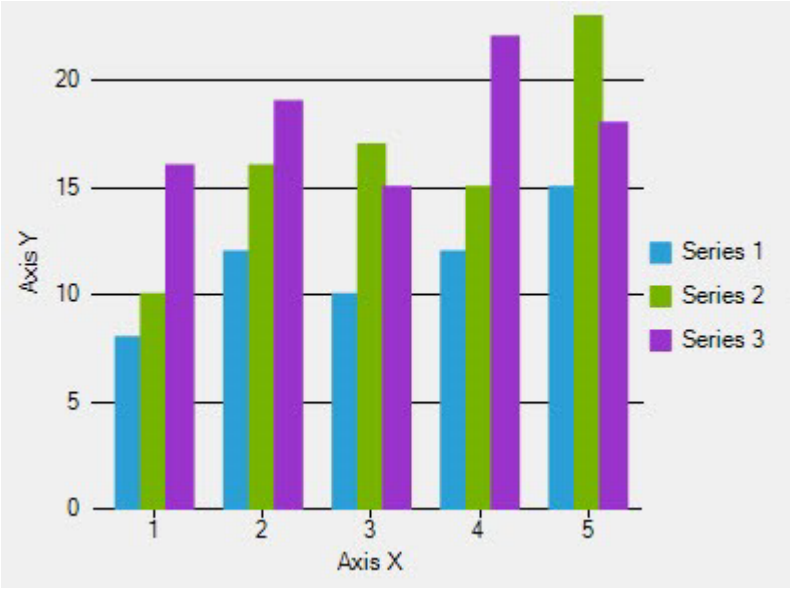
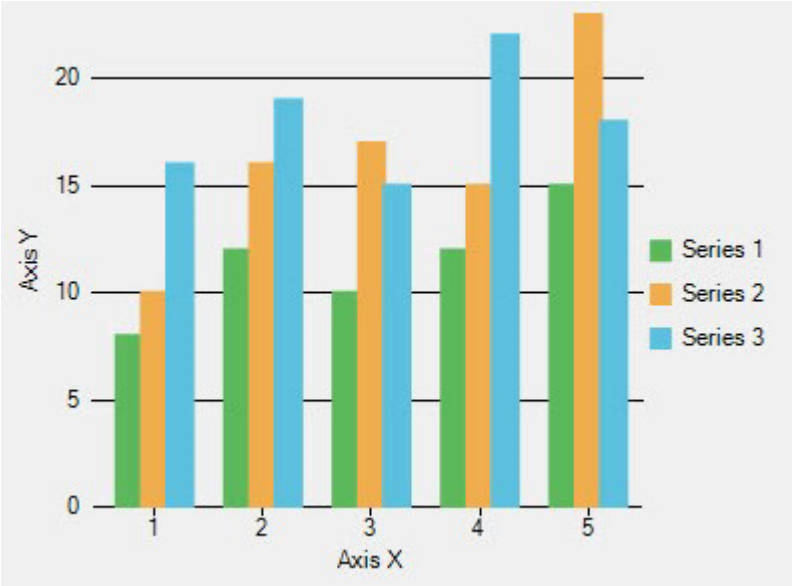
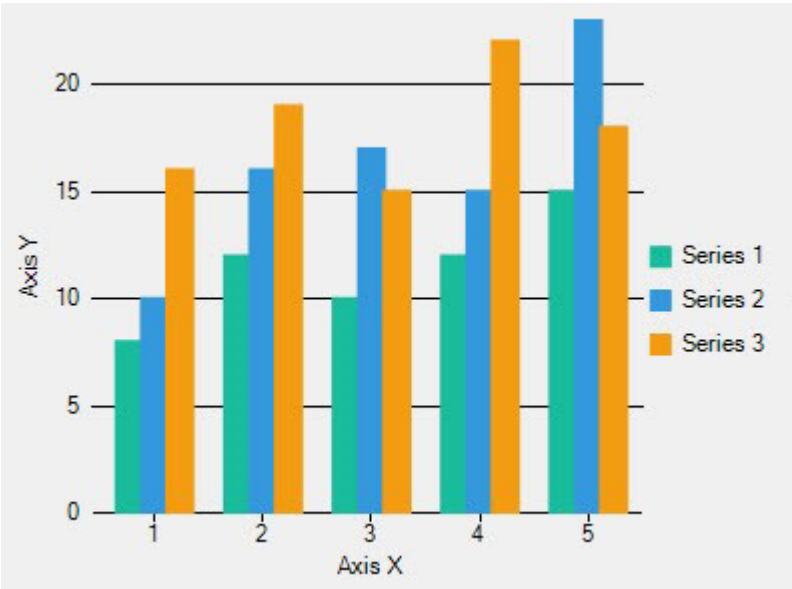
Copyright © 2019 GrapeCity, Inc. All rights reserved.

Light	
Midnight	
Modern	

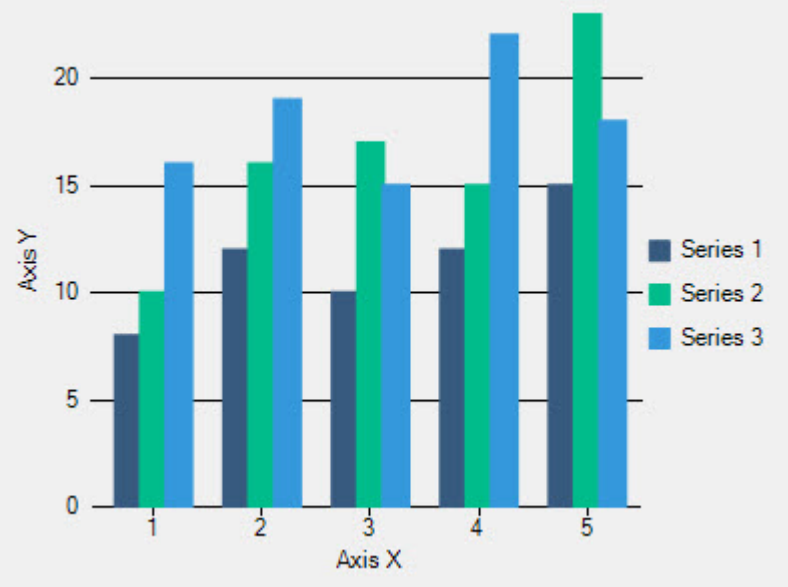
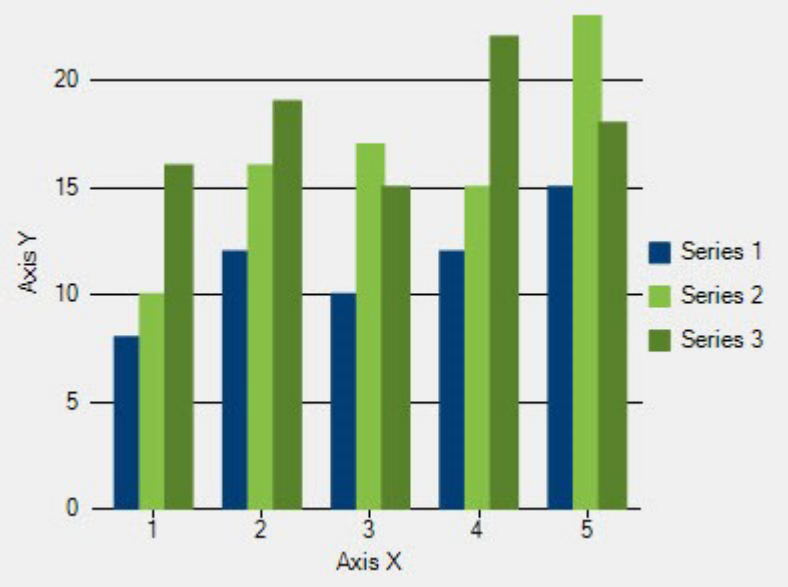
Copyright © 2019 GrapeCity, Inc. All rights reserved.

Organic	
Slate	
Zen	

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Cyborg	
Superhero	
Flatly	

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Darkly	
Cerulan	
Custom	Copies the currently specified palette into the custom group.

Specifying RGB Colors

A color can be specified by its RGB components, useful for matching another RGB color. RGB color values combine hexadecimal values for the red, green, and blue components of a color. "00" is the smallest value a component can have; "ff" is the largest value. For example, "#ff00ff" specifies magenta (the maximum value of red and blue combined with no green).

Specifying Hue, Saturation, and Brightness

In addition to a color being specified by its RGB components, it can also be represented by its hue, saturation, and brightness. The hue, saturation, and brightness are all aspects of the red, green, and blue color scheme. The hue is the specific tone of the color wheel made up of red, green, and blue tones. The saturation is the intensity of the hue from

Copyright © 2019 GrapeCity, Inc. All rights reserved.

gray tone to a pure vivid tone. And the brightness is the lightness or darkness of a tone.

Using Transparent Colors

The background and foreground of all elements except the chart itself can be "Transparent".

When a background or foreground is transparent, the chart uses the color of the element outside it for the background. For example, the header would have the background of the chart itself when its background is set to Transparent.

In other words, if the background color of the element is transparent then its background is not drawn. If the foreground color of the element is transparent, then the foreground (for example, the text of a title) is not drawn.

The transparent color properties are located under the Style nodes, found at design time on the Control, Header, Footer, Legend, ChartArea, and ChartLabels objects in the Visual Studio Properties window.

Fonts

Fonts, when customized with respect to various chart elements, enhance the impact of the chart. You can adjust the font size of an element to make it more suitable as per the overall size of the chart.

To change or customize fonts in **FlexChart**, you can use the following properties provided by the ChartStyle object:

Property	Description
FontFamily	Sets the font family.
FontSize	Sets the font size.
FontStretch	Sets the font stretch.
FontStyle	Sets the font style.
FontWeight	Sets the font weight.

Symbol Styles for Series

Depending upon the requirements, you may need to customize the appearance of series in the chart.

FlexChart allows you to customize series in the chart with the SymbolMarker and the SymbolSize property.

The **SymbolMarker** property allows you to set the shape of the marker to be used for each data point in the series. The **SymbolSize** property enables you to set the size (in pixels) of the symbols used to render the series.

Below is a table that lists how these properties affect each chart type:

Value	Effect for SymbolMarker	Effect for SymbolSize
ChartType.Column	No effect	No effect
ChartType.Bar	No effect	No effect
ChartType.Line	No effect	No effect
ChartType.Scatter	Changes the symbol marker	Changes the symbol size

Copyright © 2019 GrapeCity, Inc. All rights reserved.

ChartType.LineSymbols	Changes the symbol marker	Changes the symbol size
ChartType.Area	No effect	No effect
ChartType.Spline	No effect	No effect
ChartType.SplineSymbols	Changes the symbol marker	Changes the symbol size
ChartType.SplineArea	No effect	No effect
ChartType.Bubble	Changes the symbol marker	No effect
ChartType.Candlestick	No effect	Changes the symbol size
ChartType.HighLowOpenClose	No effect	Changes the symbol size

The SymbolSize property has no effect on the Bubble Chart; however, you can change the size of the bubble in the Bubble Chart by setting the BubbleMaxSize and the BubbleMinSize property located in the Options node of the Properties window.

End-User Interaction

When it comes to the functionality and features of the chart, you may have specific requirements that can be accommodated only through a few specific tools.

Therefore, to accommodate such requirements, **FlexChart** renders a set of conversion methods and interactive built-in tools. These tools help you customize and develop your applications further.

Go to the following sections for information on end-user interaction:

- ToolTips
- Axis Scrollbar
- Range Selector
- Line Marker
- Hit test

ToolTips

Tooltips are pop-ups that appear while hovering over data points or series in a chart. They provide additional, valuable information about chart data in scenarios, as follows:

- **Single series chart**: Tooltips display data values and series name.
- **Mixed charts**: Tooltips display multiple data values for multiple series for a single category.
- **Pie charts**: Tooltips display name and percentage share or value of slices.

A tooltip displays Y value of a data point in FlexChart by default. However, FlexChart allows creating and formatting custom content in tooltips using pre-defined parameters and formats. In addition, the control allows creating a shared tooltip in the case of mixed charts.

To know more about tooltips in FlexChart, see the following topics:

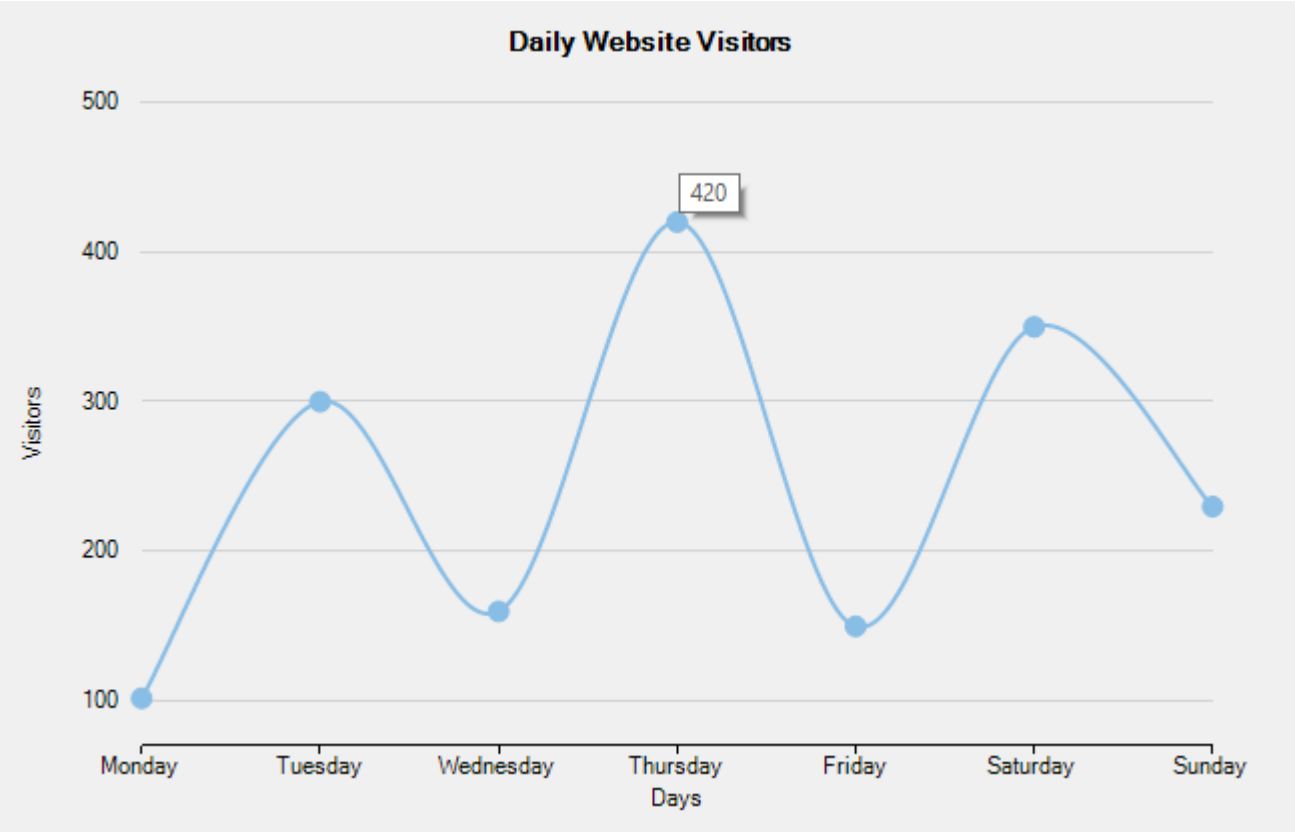
- Default Tooltip
- Customizing Tooltip Content

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Default Tooltip

FlexChart displays a default tooltip when you hover over a data point or series. The default tooltip shows the Y value of the hovered data point. FlexChart generates the default tooltip using the underlying data when custom content is not present.

The following image displays the default tooltip showing the data value of a data point.



Customizing Tooltip Content

FlexChart simplifies customizing tooltip content by allowing you to set pre-defined parameters in the ToolTipContent property of the tooltip.

To customize content in a tooltip, set the pre-defined parameters in the template string of the ToolTipContent property of the FlexChartBase class.

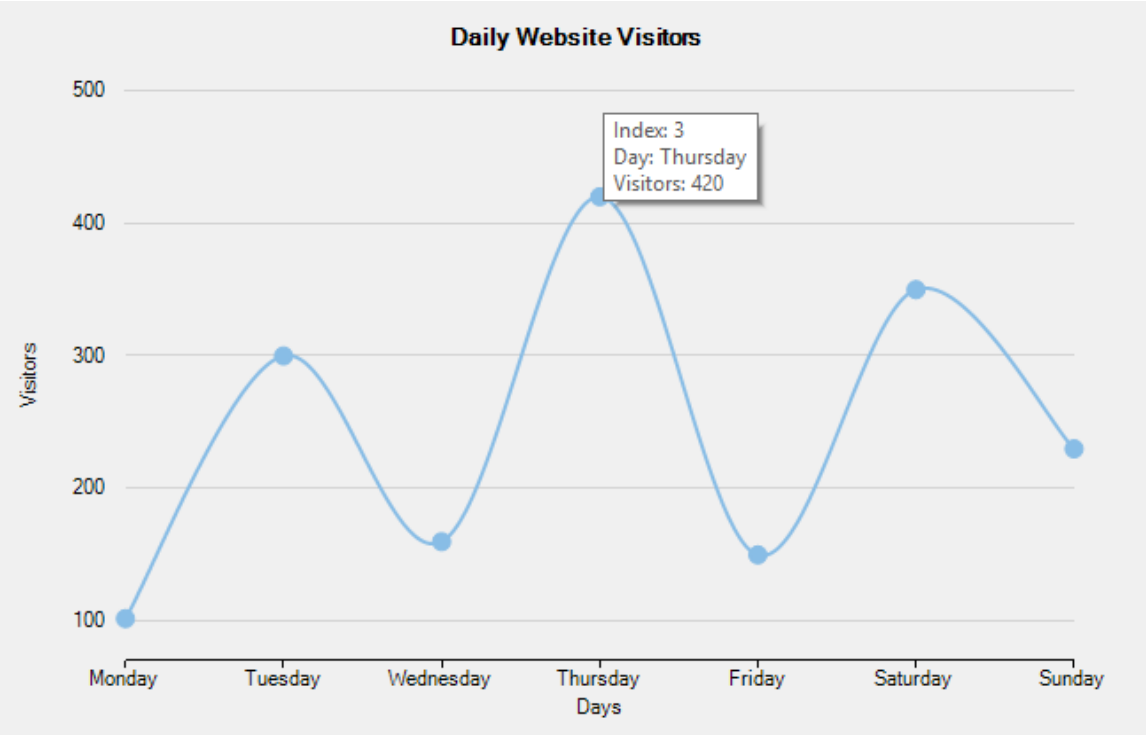
The table below lists the pre-defined parameters applicable for tooltip content customization.

Parameter	Description
x	Shows the X value of the data point.
y	Shows the Y value of the data point.
value	Shows the Y value of the data point.
name	Shows the X value of the data point.
seriesName	Shows the name of the series.

Copyright © 2019 GrapeCity, Inc. All rights reserved.

pointIndex	Shows the index of the data point.

The following image displays customized tooltip content showing the index and the data point values.



The following code compares and displays data of number of daily website visitors in a specific week. The code shows how to configure the ToolTipContent property to customize tooltip content.

XAML

```
<c1:C1FlexChart Name="flexChart"
                ToolTipContent="Index: {pointIndex}&#13;Day: {name}&#13;{seriesName}: {Visitors}">
</c1:C1FlexChart>
```

Formatting Tooltip Content

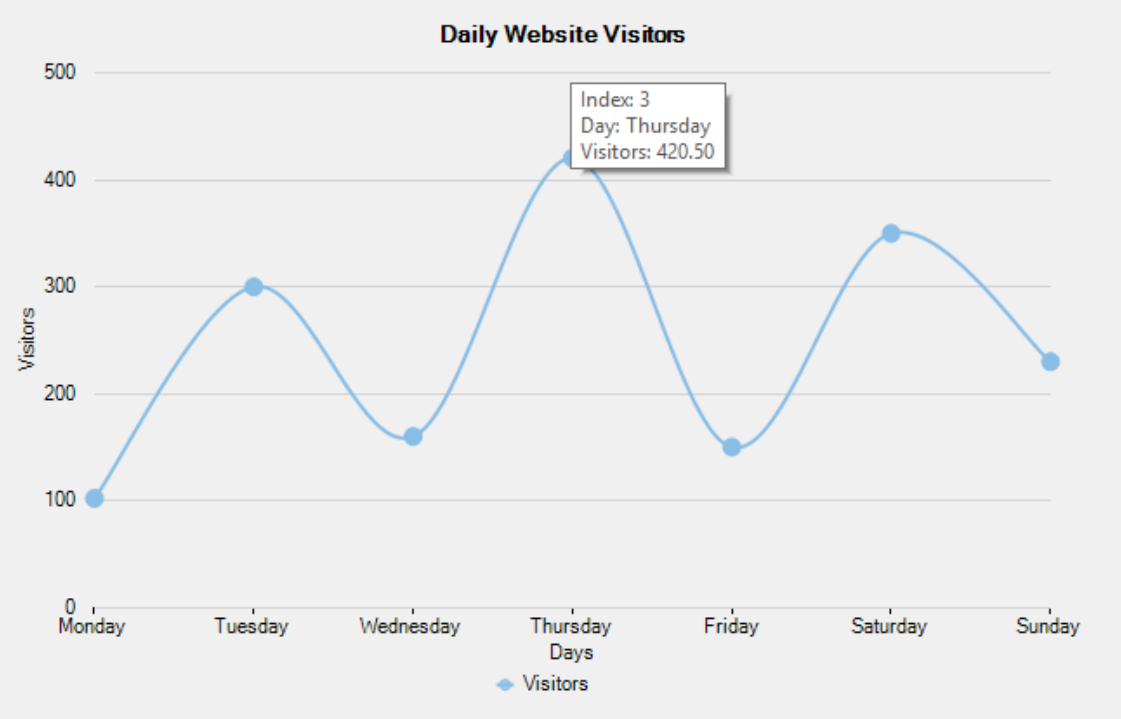
In FlexChart, it is possible to display number separators, current symbols, or date/time formats to add more details into tooltips.

FlexChart enables you to format the custom content in the tooltip by using standard and custom format strings. These format strings are a variety of Numeric and DateTime formats provided by .NET.

For information about these format strings, refer to Numeric and DateTime format strings.

The following image displays customized tooltip content showing the index and formatted values of the data point.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



The following code compares and displays data of number of daily website visitors in a specific week. The code shows how to configure the ToolTipContent property to format tooltip content.

XAML

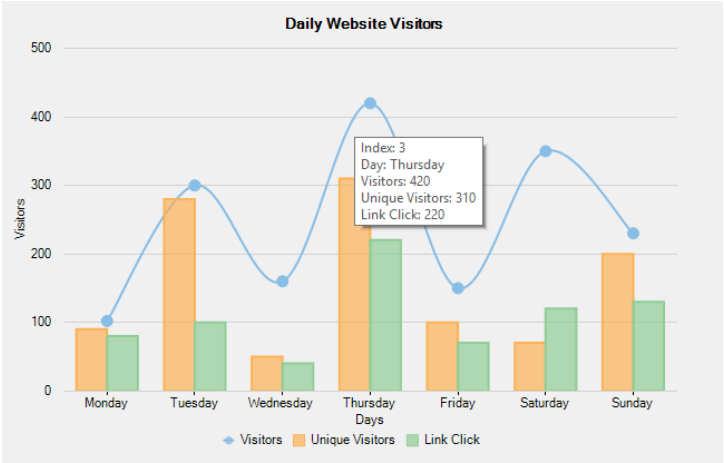
```
<c1:C1FlexChart Name="flexChart"
                ToolTipContent="Index: {pointIndex}&#13;Day: {name}&#13;{seriesName}: {Visitors:F}">
</c1:C1FlexChart>
```

Shared Tooltip

A shared tooltip is a single tooltip that highlights all data values for a single X value in the chart.

In a chart containing mixed chart types, you often require displaying multiple Y values for a common X value through a single tooltip. In such cases, FlexChart tooltips can be used as shared tooltips by setting the ToolTipContent property accordingly.

The following image displays a shared tooltip showing Y values for all series at a single X value.



The code compares and displays data of number of daily website visitors, unique visitors, and link clicks in a specific week. The code shows how to set the ToolTipContent property to create a shared tooltip.

XAML

```
<c1:C1FlexChart Name="flexChart"
            ToolTipContent="Index: {pointIndex}&#13;Day: {name}&#13;Visitors: {Visitors}&#13;Unique Visitors: {UniqueVisitors}&#13;Link Click: {LinkClick}">
</c1:C1FlexChart>
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Axis Scrollbar

The presence of a large number of values or data in charts makes data interpretation difficult, especially in compact user interfaces. Axis Scrollbars solve this problem by letting you easily interpret closely related data within a specific range.

FlexChart allows you to add Axis Scrollbar to primary axes (X and Y axes) as well as secondary axes. To add Axis Scrollbar to an axis, you need to create an instance of the C1.WPF.Chart.Interaction.C1AxisScrollbar class.

The C1AxisScrollbar class provides the ScrollButtonsVisible property that accepts Boolean values to set the visibility of the scrollbar buttons. To set the current lower and the current upper magnitude of the scrollbar, you can use the LowerValue and the UpperValue property provided by C1RangeSlider class respectively. The lower and upper values change when the scrollbar is resized or moved. When any of the LowerValue or the UpperValue property changes, the ValueChanged event provided by the C1RangeSlider class fires.

See the following code snippet for reference:

XAML

```
<c1:Axis.Scrollbar>
    <c1:C1AxisScrollbar x:Name="axisYScrollbar" ScrollButtonsVisible="False" Width="30"/>
</c1:Axis.Scrollbar>
```

Code

C# copyCode

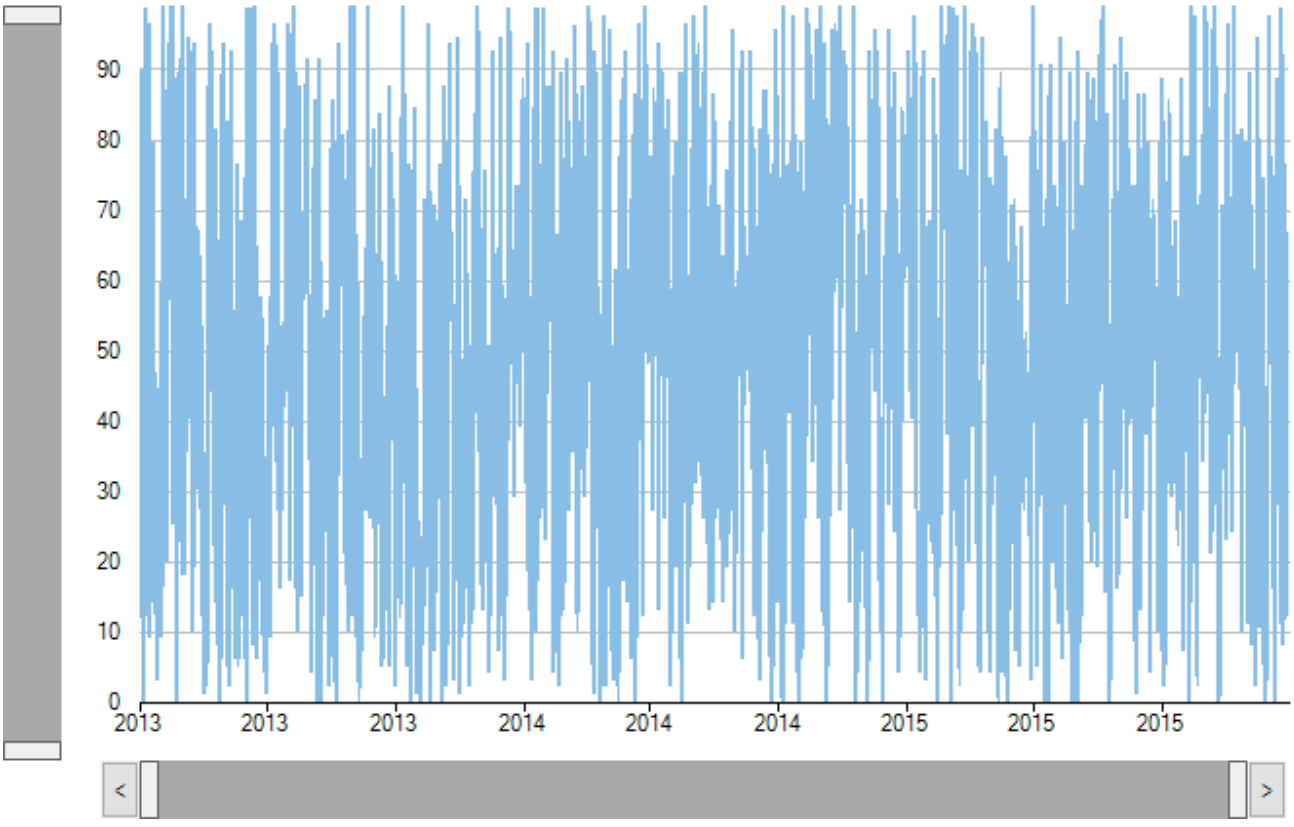
```
public class AxisScrollbarModel
{
    Random rnd = new Random();

    public List<DataItem> Data
    {
        get
        {
            var pointsCount = rnd.Next(1, 30);
            var pointsList = new List<DataItem>();
            for (DateTime date = new DateTime(DateTime.Now.Year - 3, 1, 1); date.Year <
DateTime.Now.Year; date = date.AddDays(1))
            {
                pointsList.Add(new DataItem()
                {
                    Date = date,
                    Series1 = rnd.Next(100)
                });
            }

            return pointsList;
        }
    }
}
```

VB copyCode

```
Public Class AxisScrollbarModel
    Private rnd As New Random()
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
    Public ReadOnly Property Data() As List(Of DataItem)
        Get
            Dim pointsCount = rnd.[Next](1, 30)
            Dim pointsList = New List(Of DataItem)()
            Dim [date] As New DateTime(DateTime.Now.Year - 3, 1, 1)
            While [date].Year < DateTime.Now.Year
                pointsList.Add(New DataItem() With {
                    .[Date] = [date],
                    .Series1 = rnd.[Next](100)
                })
                [date] = [date].AddDays(1)
            End While

            Return pointsList
        End Get
    End Property
End Class
```



Range Selector

While scrollbars are a traditional way of scrolling the chart, Range Selector is a more modern approach, which lets the user visualize where the selected range sits in the complete data range.

FlexChart's Range Selector lets you select a range of numeric data with lower value thumb and upper value thumb. These thumbs define the start and end values of the range. On dragging the thumb towards left (or down) on the range bar, you reduce its value, and dragging it towards the right (or up) increases the value on the range bar.

Copyright © 2019 GrapeCity, Inc. All rights reserved.

In FlexChart, you can work with Range Selector by creating an instance of the C1.WPF.Chart.Interaction.C1RangeSelector class and adding it to the Layers collection of the chart by using the Layers property of C1FlexChart. The C1RangeSelector class inherits the C1.WPF.C1RangeSlider class. You can use the LowerValue and the UpperValue property provided by C1RangeSlider to set the lower and the upper value of the range selector respectively. The ValueChanged event fires when any of the LowerValue or the UpperValue property is changed.

To set the horizontal or the vertical orientation of the range selector, you can use the Orientation property. When the property is changed, the OrientationChanged event fires.

Here is the code snippet showing the implementation:

XAML

```
<c1:C1FlexChart.Layers>
    <c1:C1RangeSelector x:Name="rangeSelector" Grid.Row="4" Margin="110 10 0 10"/>
</c1:C1FlexChart.Layers>
```

Code

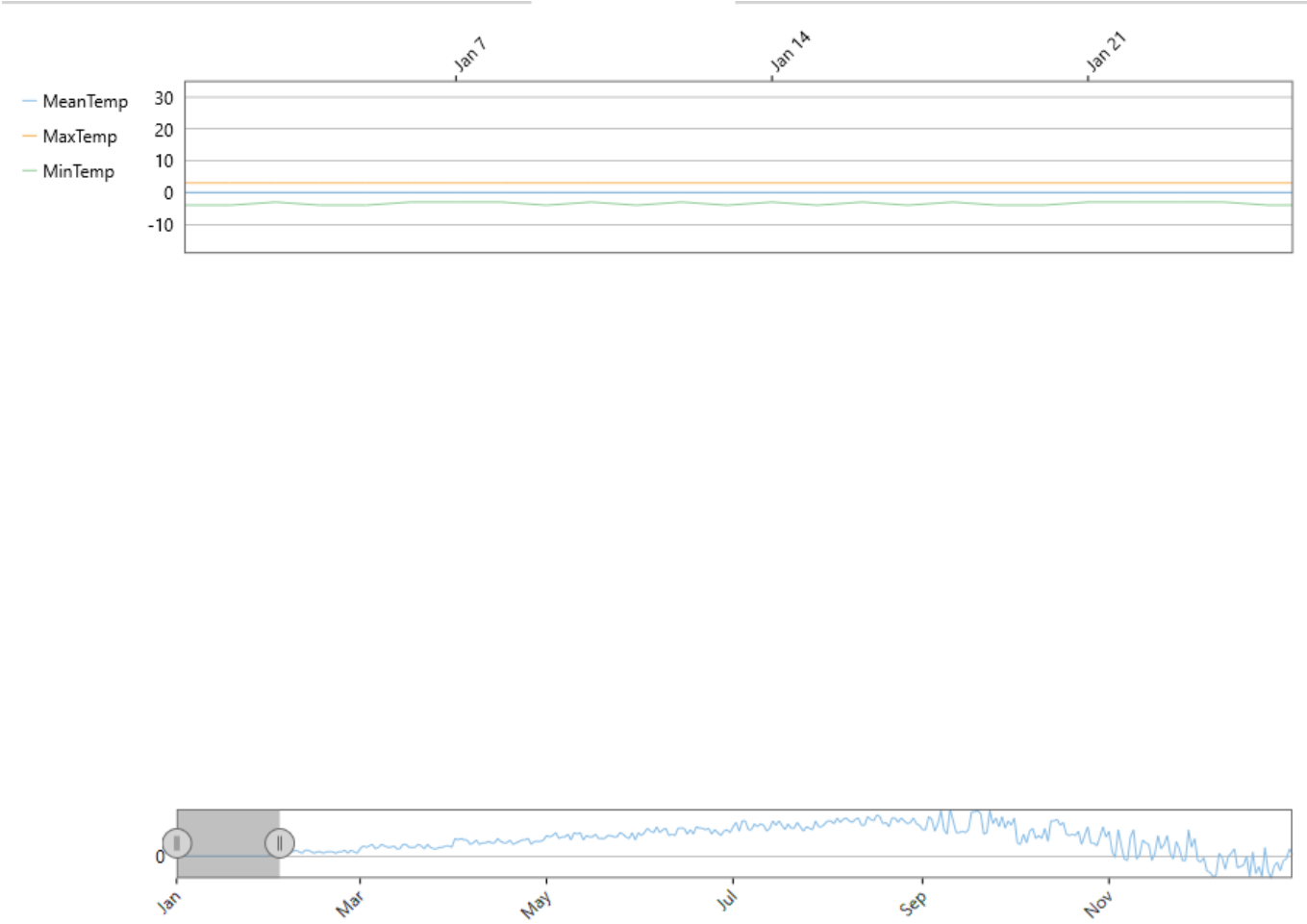
C# copyCode

```
void OnChartRendered(object sender, RenderEventArgs e)
{
    var flexChart = sender as C1FlexChart;
    if (flexChart == null)
        return;

    var rect = flexChart.PlotRect;
    e.Engine.SetFill(Colors.Transparent);
    e.Engine.SetStroke(new SolidColorBrush(Colors.DimGray));
    e.Engine.SetStrokeThickness(1d);
    e.Engine.DrawRect(rect.X, rect.Y, rect.Width, rect.Height);
}
```

VB copyCode

```
Private Sub OnChartRendered(sender As Object, e As RenderEventArgs)
    Dim flexChart = TryCast(sender, C1FlexChart)
    If flexChart Is Nothing Then
        Return
    End If

    Dim rect = flexChart.PlotRect
    e.Engine.SetFill(Colors.Transparent)
    e.Engine.SetStroke(New SolidColorBrush(Colors.DimGray))
    e.Engine.SetStrokeThickness(1.0)
    e.Engine.DrawRect(rect.X, rect.Y, rect.Width, rect.Height)
End Sub
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.



Line Marker

LineMarker displays the precise data values for a given position on the chart by dragging horizontal and/or vertical lines over the plot with an attached label. It is useful in scenarios, where a user has a lot of data in a line or area chart, or if a user wants to display data from multiple series in a single label. With built-in interactions, such as Drag and Move. a user can drag the line marker and more precisely select the data point on the chart.

To create a line marker and use it in FlexChart, you need to create an instance of the C1.WPF.Chart.Interaction.C1LineMarker class and add it to the Layers collection of the chart by using the Layers property of C1FlexChart.

You need to use the Lines property provided by C1LineMarker to set the visibility of the LineMarker lines. The Lines property accepts the following values from the LineMarkerLines enumeration:

- **Both**: Shows both vertical and horizontal lines
- **Horizontal**: Shows a horizontal line
- **Vertical**: Shows a vertical line
- **None**: Shows no line

The C1LineMarker class also provides the Alignment property to set the alignment of the line marker. In addition, you can set the interaction mode of the line marker by setting the Interaction property to any of the following values in the LineMarkerInteraction enumeration:

- **Drag**: The line marker moves when the user drags the line
- **Move** (Default): The line marker moves with the pointer

Copyright © 2019 GrapeCity, Inc. All rights reserved.

- **None**: The user specifies the position by clicking.

If you set the Interaction property to Drag, you need to set the DragContent and the DragLines property to specify whether the content and values linked with the line marker lines are draggable or not. Furthermore, you can set the initial position of the line marker relative to the plot area with the help of VerticalPosition and HorizontalPostion properties. The acceptable range for these properties is [0,1].

Below is the code snippet with the implementation.

XAML

```
<Chart:C1FlexChart.Layers>
    <Chart:C1LineMarker x:Name="lineMarker" Lines="Vertical"
    Grid.Row="2" PositionChanged="OnLineMarkerPositionChanged"
    VerticalAlignment="Top"/>
</Chart:C1FlexChart.Layers>
```

Code

C# copyCode

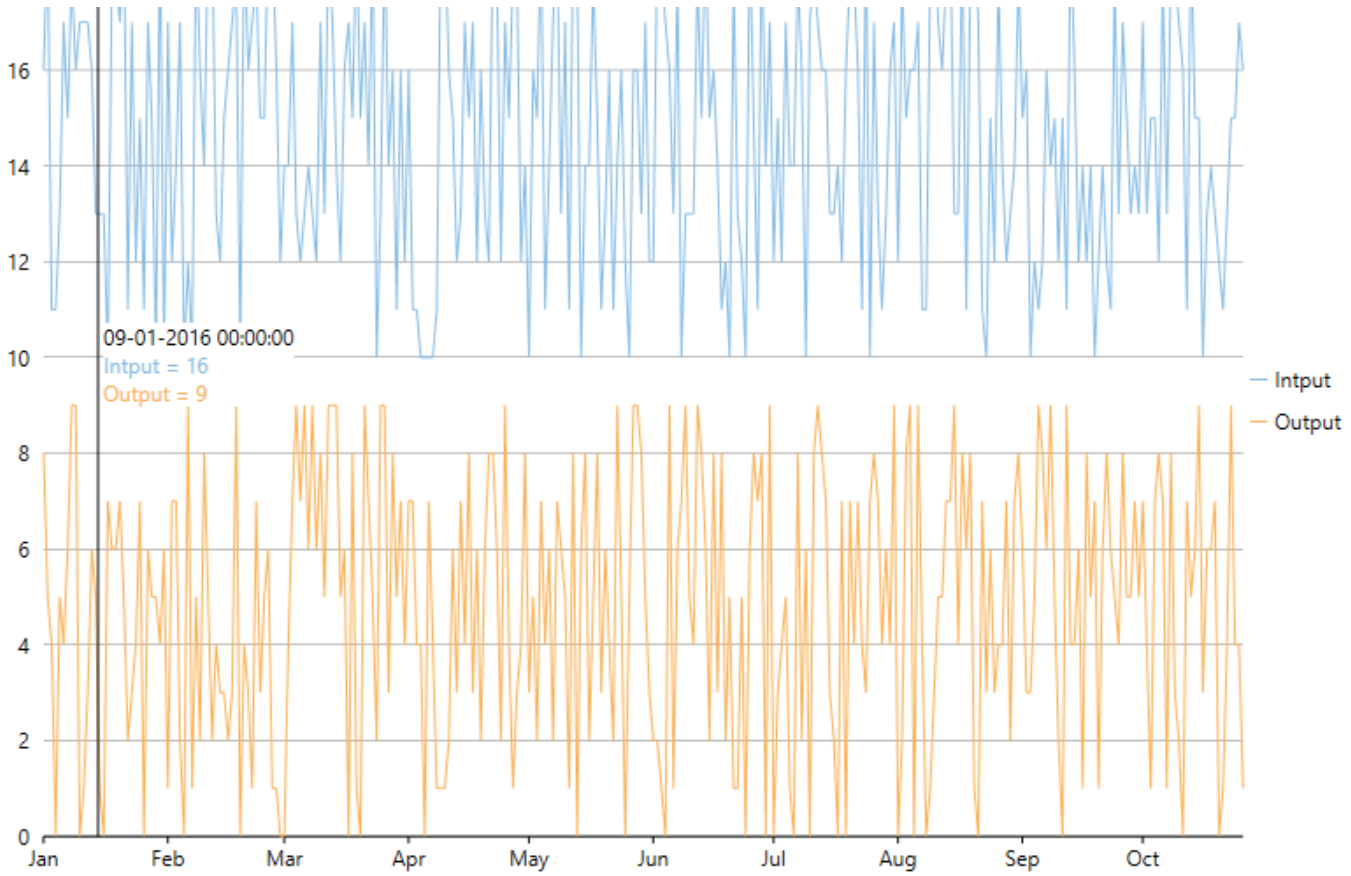
```
private void OnLineMarkerPositionChanged(object sender, PositionChangedArgs e)
{
    if (flexChart != null)
    {
        var info = flexChart.HitTest(e.Position);
        int pointIndex = info.PointIndex;
        var tb = new TextBlock();
        tb.Inlines.Add(new Run()
        {
            Text = info.X.ToString()
        });
        for (int index = 0; index < flexChart.Series.Count; index++)
        {
            var series = flexChart.Series[index];
            var value = series.GetValues(0)[pointIndex];
            var fill = (int)((IChart)flexChart).GetColor(index);
            string content = string.Format("{0}{1} = {2}", "\n",
            series.SeriesName, value.ToString());
            tb.Inlines.Add(new Run()
            {
                Text = content,
                Foreground = new SolidColorBrush() { Color = FromArgb(fill) }
            });
        }
        lineMarker.Content = tb;
    }
}
```

VB copyCode

```
    Private Sub OnLineMarkerPositionChanged(sender As Object, e As
PositionChangedArgs)
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

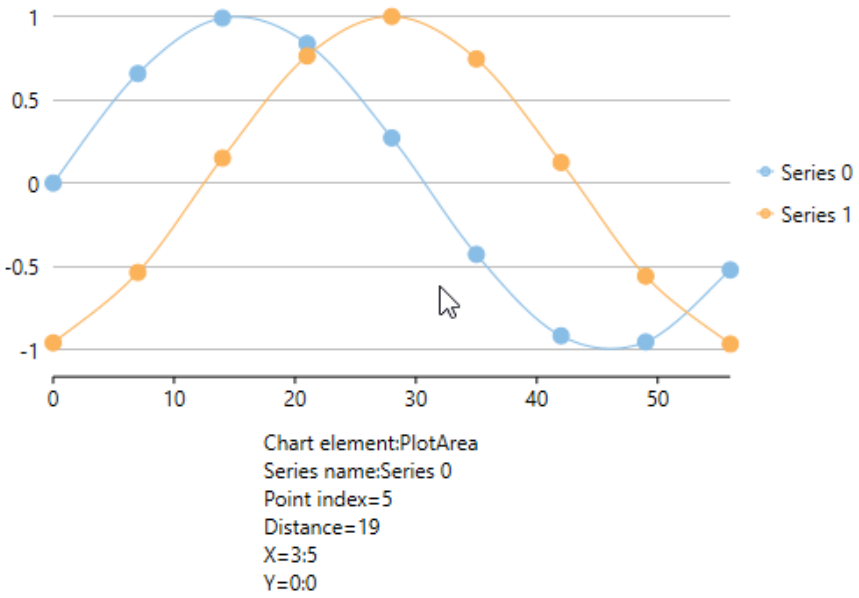
```
        If flexChart IsNot Nothing Then
            Dim info = flexChart.HitTest(e.Position)
            Dim pointIndex As Integer = info.PointIndex
            Dim tb = New TextBlock()
            tb.Inlines.Add(New Run() With {
                .Text = info.X.ToString()
            })
            For index As Integer = 0 To flexChart.Series.Count - 1
                Dim series = flexChart.Series(index)
                Dim value = series.GetValues(0)(pointIndex)
                Dim fill = CInt(DirectCast(flexChart, IChart).GetColor(index))
                Dim content As String = String.Format("{0}{1} = {2}",
vbLf, series.SeriesName, value.ToString())
                tb.Inlines.Add(New Run() With {
                    .Text = content,
                    .Foreground = New SolidColorBrush() With {
                    .Color = FromArgb(fill)
                    }
                })
            Next
            lineMarker.Content = tb
        End If
    End Sub
```



Copyright © 2019 GrapeCity, Inc. All rights reserved.

Hit Test

FlexChart supports hit testing, which enables you to fetch information about a specific point in the control at run-time. The information obtained about the pointed coordinate can then be reused to drill down the chart data, to set alerts, or to enable other user interaction functionalities.



FlexChart supports hit testing by utilizing HitTest() method. This method takes the location (coordinates) of the pointed entity; and returns an object of HitTestInfo class, which provides the following information about the pointer location:

- Chart element comprising the pointer
- Distance of the pointer location from the closest data point in chart, if the pointer is within plot area. Distance is returned as Double.Nan, if the pointer is outside the plot area.
- Data object corresponding to the closest data point
- Index of the nearest data point
- Series name that the nearest data point belongs to
- X value of the nearest data point
- Y value of the nearest data point

Note that, the mouse coordinates that are passed to **HitTest()** method are in pixels and are relative to the upper left corner of the window.

In this example, **HitTest()** method is called on MouseMove event of the FlexChart control. Here, the point coordinates of pointer location are passed as parameter to HitTest() method.

To enable hit testing in FlexChart, follow these steps:

1. **Add a data bound FlexChart control**
2. **Subscribe to a Mouse event**
3. **Invoke chart's HitTest method in mouse event handler**
4. **Use the information returned by HitTestInfo object**

Back to Top

1. **Add a data bound FlexChart control**

   Add an instance of FlexChart control to your WPF application, and bind it to an appropriate data source, as shown in the below code snippet.

   Xaml

```
<c1:C1FlexChart x:Name="flexChart"
                        Binding="YVals"
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
            BindingX="XVals"
            ChartType="SplineSymbols"
            Margin="10,10,0,52"
            HorizontalAlignment="Left"
            Grid.RowSpan="2"
                        Width="490">
                        <c1:Series x:Name="series0" SeriesName="Series 0"/>
                                <c1:Series x:Name="series1" SeriesName="Series
1" />
                        </c1:C1FlexChart>
```

Back to Top

2. **Subscribe to a Mouse event**

   Subscribe to a mouse event to capture the pointer coordinates, as shown in the below code snippet.

   - **Xaml**

```
 <c1:C1FlexChart Binding="YVals"
              BindingX="XVals"
              ChartType="SplineSymbols"
              x:Name="flexChart"
              MouseLeftButtonDown="flexChart_MouseLeftButtonDown"
              Margin="10,10,0,52"
              HorizontalAlignment="Left"
              Grid.RowSpan="2" Width="490">
    <c1:Series x:Name="series0" SeriesName="Series 0"/>
    <c1:Series x:Name="series1" SeriesName="Series 1" />
</c1:C1FlexChart>
```

Back to Top

3. **Invoke chart's HitTest method in mouse event handler**

   In the respective event handler, invoke the **HitTest()** method and pass the captured mouse pointer coordinates, as shown in the below code snippet.

   - **VB**

```
    Private Sub flexChart_MouseLeftButtonDown(sender As Object, e As MouseButtonEventArgs)

    HitTestOnFlexChart(e.GetPosition(flexChart))
End Sub
```

   - **C#**

```
private void flexChart_MouseLeftButtonDown(object sender, MouseButtonEventArgs e)
{
    HitTestOnFlexChart(e.GetPosition(flexChart));
}
```

Back to Top

4. **Use the information returned by HitTestInfo object**

   The information regarding mouse pointer location, as returned by the **HitTestInfo** object, can then be reused. For example in the below code snippet, the values returned by **HitTestInfo** object are converted to string and displayed in a TextBlock.

   - **VB**

```
    Private Sub HitTestOnFlexChart(p As Point)
  ' Show information about chart element under mouse/touch.
  Dim ht = flexChart.HitTest(p)

  Dim result = New StringBuilder()

  result.AppendLine(String.Format("Chart element:{0}", ht.ChartElement))
  If ht.Series IsNot Nothing Then
      result.AppendLine(String.Format("Series name:{0}", ht.Series.Name))
  End If

  If ht.PointIndex > 0 Then
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
            result.AppendLine(String.Format("Point index={0:0}", ht.PointIndex))
        End If

        If ht.Distance > 0 Then
            result.AppendLine(String.Format("Distance={0:0}", ht.Distance))
        End If

        If ht.X IsNot Nothing Then
            result.AppendLine(String.Format("X={0:0:0}", ht.X))
        End If

        If ht.Y IsNot Nothing Then
            result.AppendLine(String.Format("Y={0:0:0}", ht.Y))
        End If

        tbPosition1.Text = result.ToString()
    End Sub
```

- **C#**

```
    void HitTestOnFlexChart(Point p)
    {
        // Show information about chart element under mouse/touch.
        var ht = flexChart.HitTest(p);
        var result = new StringBuilder();
        result.AppendLine(string.Format("Chart element:{0}", ht.ChartElement));
        if (ht.Series != null)
            result.AppendLine(string.Format("Series name:{0}", ht.Series.Name));
        if (ht.PointIndex > 0)
            result.AppendLine(string.Format("Point index={0:0}", ht.PointIndex));
        if (ht.Distance > 0)
            result.AppendLine(string.Format("Distance={0:0}", ht.Distance));
        if (ht.X != null)
            result.AppendLine(string.Format("X={0:0:0}", ht.X));
        if (ht.Y != null)
            result.AppendLine(string.Format("Y={0:0:0}", ht.Y));
        tbPosition1.Text = result.ToString();
    }
```

Back to Top

Animation

FlexChart allows you to control how the animation is applied to each series and series elements. It allows you to enable chart animation effects through a combination of different properties available in the FlexChart class. These properties allow you to apply duration, delay and an easing function for each animation. The animation effects are applied in one of the two scenarios, either while loading the chart for the first time or while the chart is redrawn after modifications.

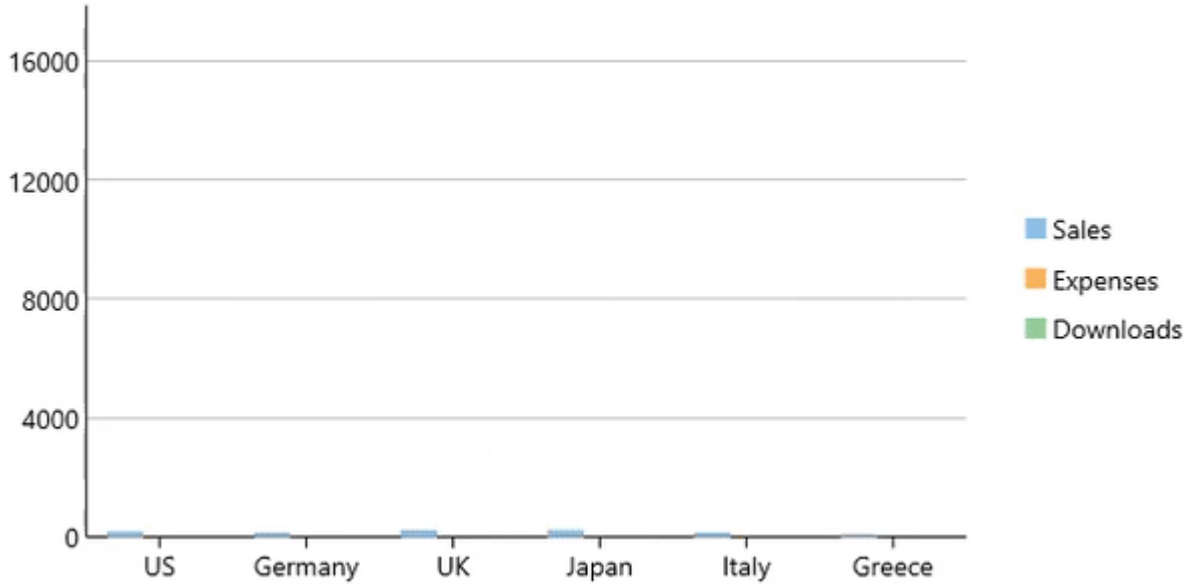
The FlexChart control supports two basic scenarios for animation.

- **Load**
- **Update**

Load

Implement animation when the data is first loaded into the chart, for example when a new series is added to the FlexChart, you can apply animation properties. The following image shows how animation works while loading the FlexChart control.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



Use the following code to implement animation while loading the FlexChart control.

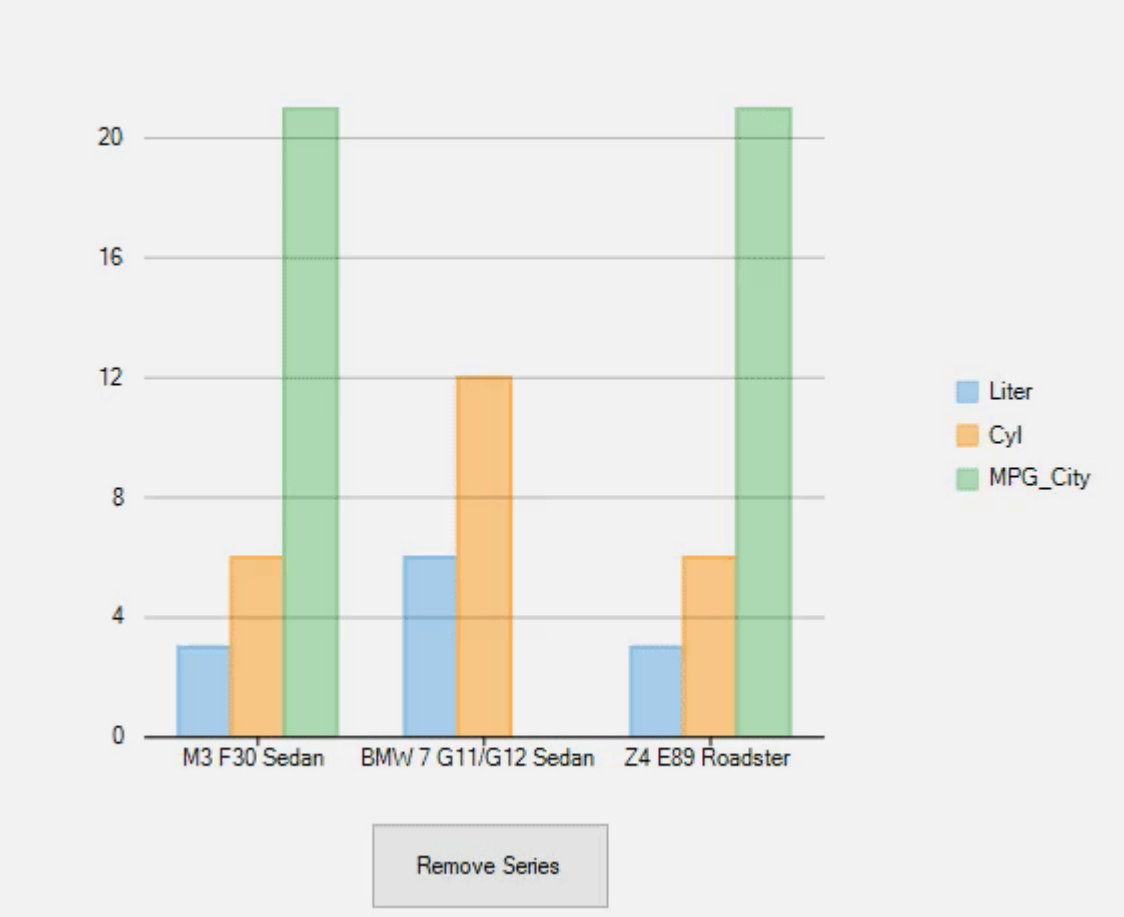
C#

```
// Animation
flexChart1.AnimationSettings = C1.Chart.AnimationSettings.Load;
flexChart1.AnimationUpdate.Easing = C1.Chart.Easing.Linear;
flexChart1.AnimationUpdate.Duration = 500;
flexChart1.AnimationLoad.Type = C1.Chart.AnimationType.Series;
```

Update

Implement animation when existing data is modified in the chart, for example when a new series is added or removed from the FlexChart control, you can apply animation properties. The following image shows how animation works while updating the FlexChart control.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



Use the following code to implement animation while loading the FlexChart control.

C#

```
// Animation
flexChart1.AnimationSettings = C1.Chart.AnimationSettings.Update;
flexChart1.AnimationUpdate.Easing = C1.Chart.Easing.Linear;
flexChart1.AnimationUpdate.Duration = 500;
flexChart1.AnimationUpdate.Type = C1.Chart.AnimationType.All;

private void button1_Click(object sender, EventArgs e)
{
   //Update FlexChart
   flexChart1.BeginUpdate();
   flexChart1.Series.RemoveAt(0);
   flexChart1.EndUpdate();
}
```

Every animation scenario has its own set of options that can be set independently in the FlexChart control. These options include various properties describing the corresponding animation.

To implement animation in the FlexChart control, you need make use of the following properties.

1. AnimationSettings -This property allows the user to apply settings for the animation. It allows the user to specify that when to apply animation in the FlexChart control. This property accepts values from the AnimationSettings enumeration provided by the FlexChart class. The AnimationSettings enumeration has special flags to control axes animation (smooth transition) so that you can enable or disable smooth axis

Copyright © 2019 GrapeCity, Inc. All rights reserved.

transition for loading or updating data.

2. AnimationOptions - The AnimationLoad and AnimationUpdate properties includes the following options.

    1. Duration -This property allows you to set the duration of animation in the FlexChart control. This property accepts an integer value which defines duration in milliseconds.

    2. Easing - This property allows the user to set different type of easing functions on the FlexChart control. This property accepts values from the Easing enumeration provided by the C1.Chart namespace.

    3. Type – This property allows you to set the animation type on the FlexChart control. This property accepts the following values from the AnimationType enumeration provided by the C1.Chart namespace.

        - **All** - All plot elements animate at once from the bottom of the plot area.
        - **Series** - Each series animates one at a time from the bottom of the plot area.
        - **Points** - The plot elements appear one at a time from left to right.

FlexChart Elements

You can customize the elements of a chart to make the chart look more professional and visually appealing.

FlexChart consists of Axes, Legend, and Titles. These elements have already been discussed briefly in FlexChart Fundamentals.

Below are the sections that focus upon the customization of these elements with respect to FlexChart.

- FlexChart Axes
- FlexChart Axes Labels
- Annotations
- FlexChart Legend
- FlexChart Series
- FlexChart Data Labels
- Multiple Plot Areas

FlexChart Axis

Charts generally have two axes for measuring and categorizing data: a vertical axis (Y-axis) and a horizontal axis (X-axis). The vertical axis is also known as value axis, and the horizontal axis is also called category axis.

Not all charts depict axes in the same manner. For instance, Scatter charts and Bubble charts depict numeric values on the vertical axis as well as the horizontal axis to represent discreet or continuous numerical data. A real-time example could be how Internet Usage (Hours per Week) is plotted against different Age Groups. Here, both the items will have numeric values, and data points will be plotted corresponding to their numeric values on X and Y axes.

Other charts, such as Line, Column, Bar, and Area display numeric values on the vertical axis and categories on the horizontal axis. A real-time example could be how Internet Usage (Hours per Week) is plotted against different regions. Here, regions will be textual categories plotted on the horizontal axis.

However, FlexChart provides great flexibility, thereby allowing you to display numeric values on both X and Y axes even in case of Bar, Line, and Area charts. Also, FlexChart doesn't require any additional settings to display different types of values.

An axis in **FlexChart** is represented by the Axis class. You can access the primary axes of FlexChart by using the AxisX and the AxisY property.

The primary X-axis is rendered horizontally at the bottom, and the primary Y-axis is rendered vertically at the left. You can, however, create exceptions to this rule by customizing the primary axes and also by using multiple axes.

Copyright © 2019 GrapeCity, Inc. All rights reserved.

While working with FlexChart, you can change the way tick marks and axes labels appear. You can even lessen the number of axes labels on X and Y axes by specifying the number of units between values on the axes. In addition, you can modify the alignment and orientation of the labels and change the format of the numbers to be depicted. You can style the axes and change their position as per your requirements as well.

The sections, which are mentioned below, explain different customizations and modifications possible with respect to the FlexChart axes.

Axis Position

FlexChart lets you change the position of the axes by using the Position property.

The **Position** property for an axis can be set to the following values in the Position enumeration:

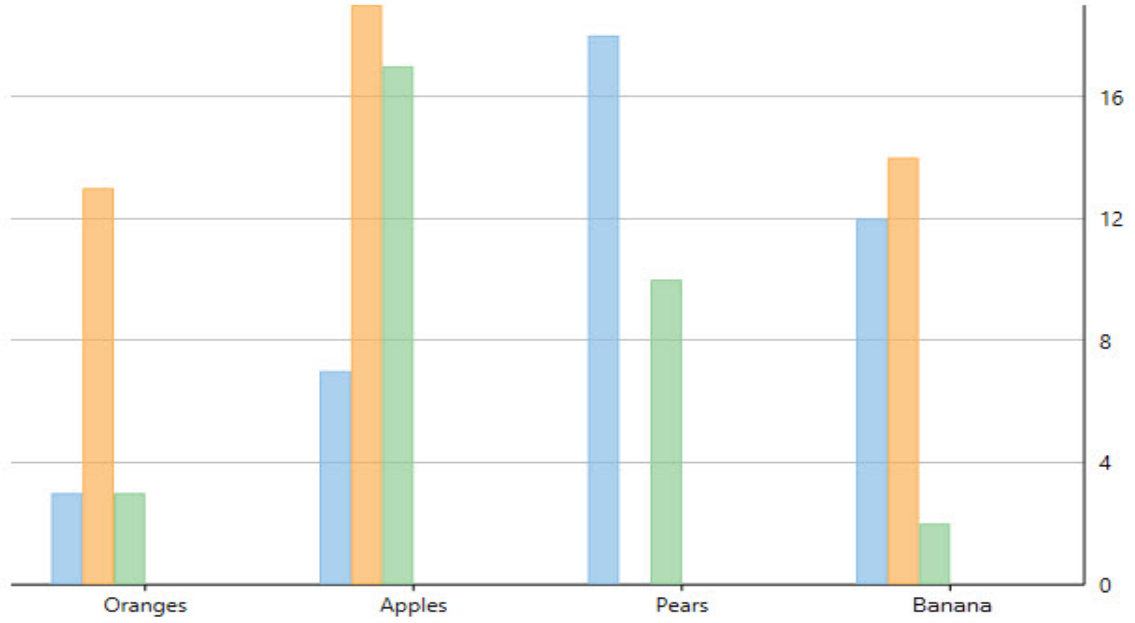
Property	Description
Position.Auto	Positions the item automatically.
Position.Bottom	Positions the item at the bottom.
Position.Left	Positions the item at the left.
Position.None	Hides the item.
Position.Right	Positions the item at the right.
Position.Top	Positions the item at the top.

Here is the sample code:

- **C#**

```
flexChart.AxisX.Position = C1.Chart.Position.Bottom;
flexChart.AxisY.Position = C1.Chart.Position.Right;
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.



Axis Title

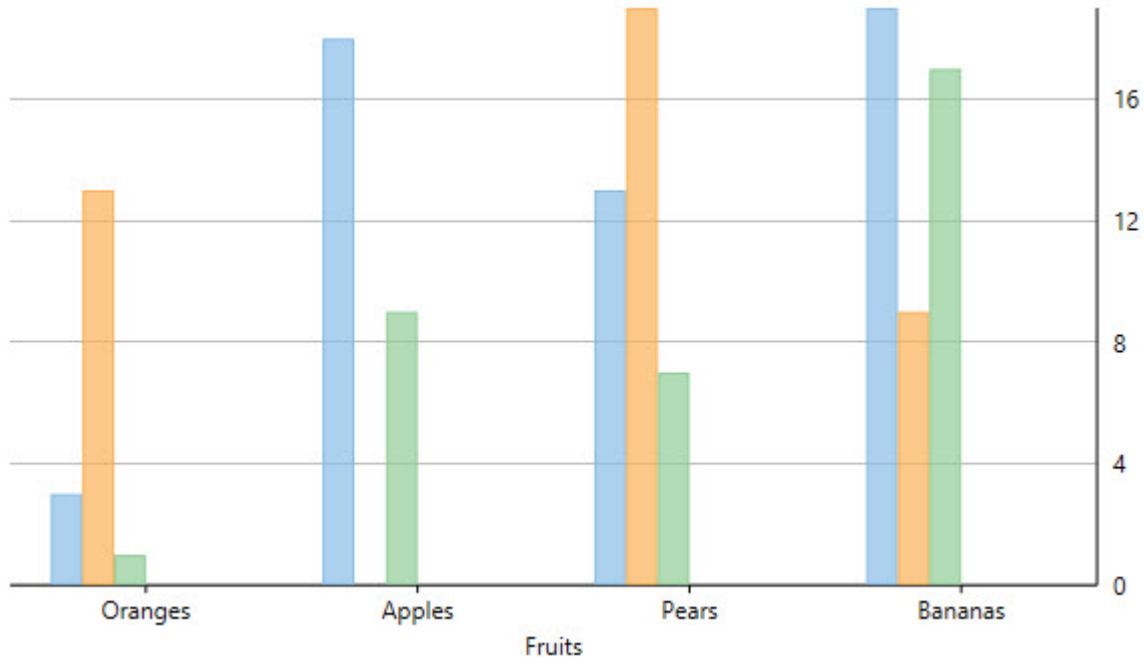
After creating a chart, you can add a title to any vertical or horizontal axis in the chart. An axis title displays information regarding what is displayed along the axis. And it enables end-users viewing the chart to understand what the data is about. It is however not possible to add axis titles to charts without axes, for instance Pie Chart.

In **FlexChart**, you can set the axis title by using the Title property, which accepts a string.

See the following code snippet:

- **C#**

```
flexChart.AxisX.Title = "Fruits";
```



Copyright © 2019 GrapeCity, Inc. All rights reserved.

Axes Tick Marks

Axes tick marks are the points at which labels are plotted on the axes. In other words, they are the small marks that identify the position of items on the axes. In addition, they divide axes into equal sections by a value determined by specific properties of an axis. And their location controls the location of grid lines.

When it comes to axes tick marks, a chart is rendered with two types basically: major tick marks and minor tick marks. Major tick marks are rendered automatically when an axis intersects the interval grid lines. And minor tick marks are rendered between major tick marks.

By default, **FlexChart** sets up X-axis with major tick marks and Y-axis with no tick marks.

You can, however, use the MajorTickMarks and the MinorTickMarks property to manipulate the position of the major tick marks and the minor tick marks respectively.

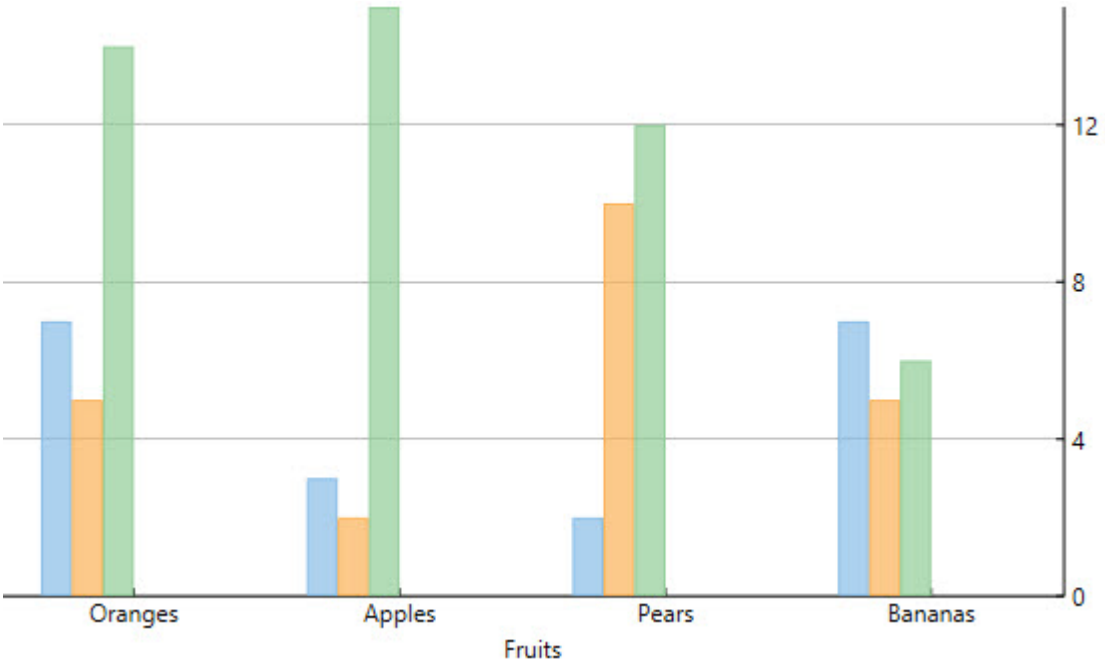
Both the properties can be set to any of the following TickMark enumeration values:

Values	Description
TickMark.Cross	Tick marks cross the axis.
TickMark.Inside	Tick marks appear inside the plot.
TickMark.None	Tick marks don't appear.
TickMark.Outside	Tick marks appear outside the plot.

See the following code sample:

- **C#**

```
flexChart.AxisX.MajorTickMarks = C1.Chart.TickMark.Inside;
flexChart.AxisY.MajorTickMarks = C1.Chart.TickMark.Inside;
```



Copyright © 2019 GrapeCity, Inc. All rights reserved.

Axis Grid Lines

Axes grid lines extend from any vertical or horizontal axis across the plot area of the chart. They are displayed for major and minor units and aligned with major and minor tick marks displayed on the axes. These auxiliary lines form a grid that improves the readability of the chart, especially when you are looking for exact values.

Primarily, axes grid lines are of two types: major grid lines and minor grid lines. The lines perpendicular to major tick marks at major unit intervals are major gridlines, while those perpendicular to minor tick marks at minor unit intervals are minor grid lines.

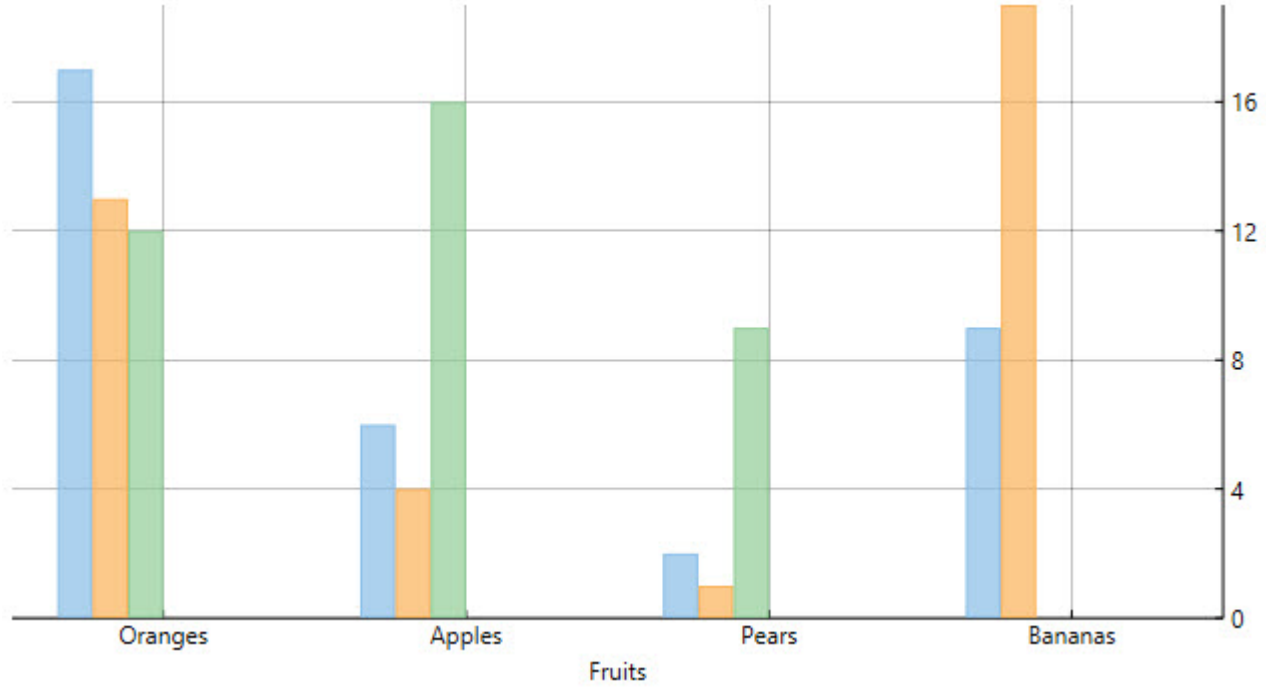
In **FlexChart**, major grid lines are controlled by the MajorGrid property, while minor grid lines are controlled by the MinorGrid property. In addition, the appearances of the major and the minor grid lines are controlled by the MajorGridStyle and the MinorGridStyle property respectively.

Using these properties, you can display horizontal as well as vertical grid lines to make the FlexChart data easier to read.

The code below illustrates how to set these properties.

- **C#**

```
flexChart.AxisY.MajorGrid = true;
flexChart.AxisX.MajorGrid = true;
```



Axis Bounds

If you want to display a specific portion of the chart in terms of data, you can do so by fixing the axes bounds. With axes bounds, the chart determines the extent of each axis by reckoning the lowest and the highest data values.

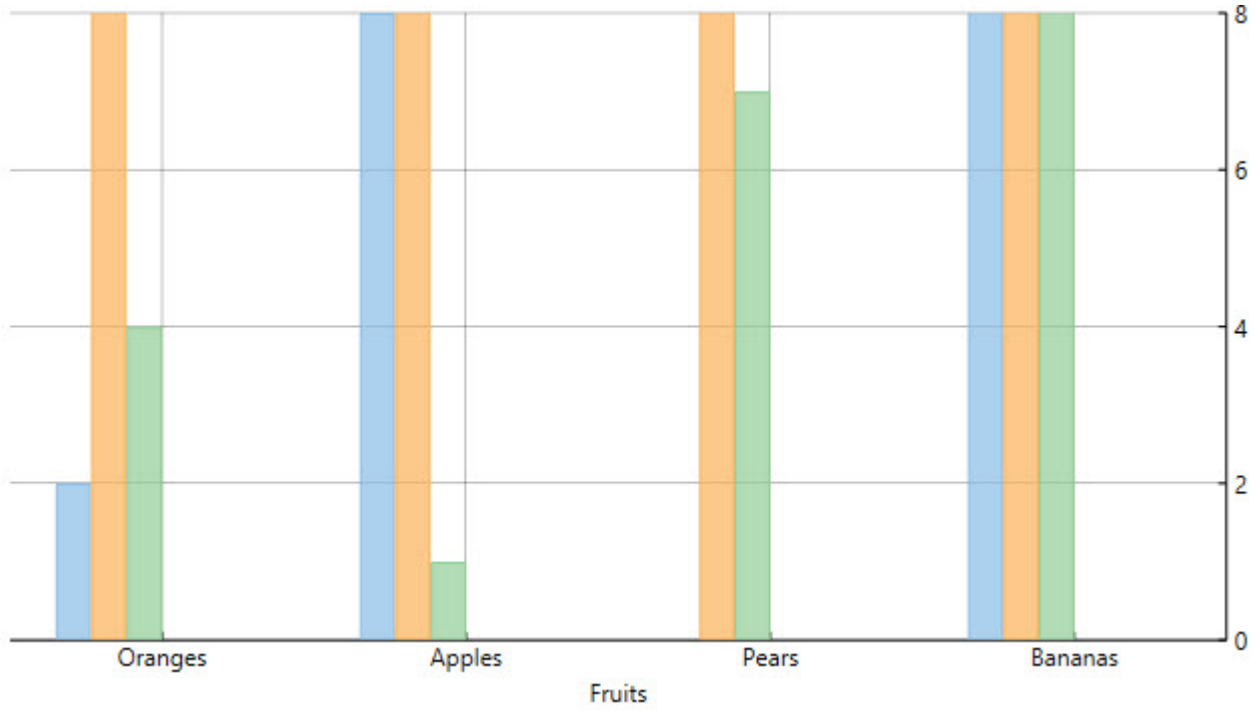
FlexChart enables you to set axes bounds by setting the Min and the Max property for the axes.

The following code shows how to set the **Min** and the **Max** property:

- **C#**

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
flexChart.AxisY.Min = 0;
flexChart.AxisY.Max = 8;
```



Axes Scaling

Sometimes, you require distinguishing the data plotted by the series in the chart. The need arises when the data points of the series do not fall in the same range. In other words, the Y axes of the series contain values in different ranges. For instance, there could be two series. The Y values for one might lie between 0 and 100 and that for the other between 0 and -100. In addition, the data of the series could require different scales altogether. In such cases, displaying the Y values of the series on a single Y-axis can confuse the interpretation of the data and overlap the same as well.

FlexChart allows you to deal with such cases by letting you scale the axes (primary and additional) by using the Min, the Max, and the MajorUnit properties. You can even apply these properties to an additional Y-axis for plotting the Y values for one of the series for better data representation.

Axis Reversing

When a dataset contains X or Y values that lie in a large range, the general chart setup sometimes doesn't display the information most effectively. Often, the chart data may look more appealing with the axes reversed.

You can reverse the axes in **FlexChart** by using the Reversed property.

Setting the **Reversed** property for the axes to True reverses the axes. This means that the maximum value along the axis takes the place of the minimum value, and the minimum value along the axis takes the place of the maximum value.

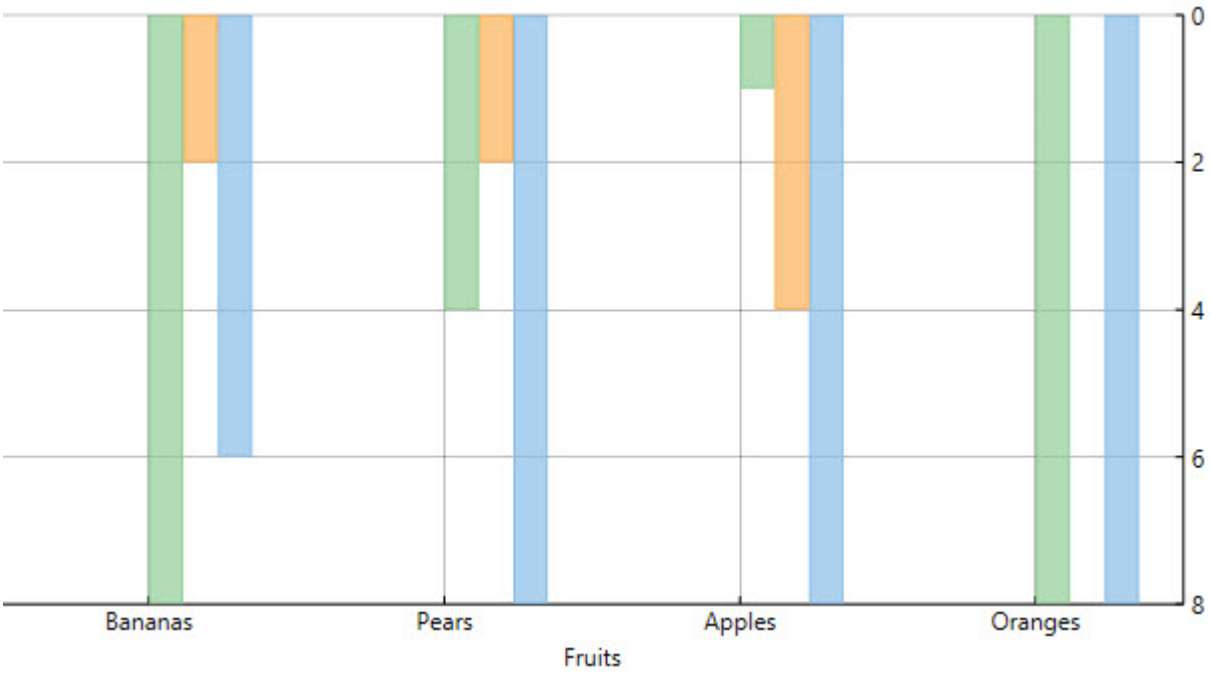
Initially, the chart displays the minimum value on the left of X-axis, and at the bottom of Y-axis. However, the Reversed property for the axes juxtaposes the maximum and minimum values.

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Here is the sample code:

- **C#**

```
flexChart.AxisX.Reversed = true;
flexChart.AxisY.Reversed = true;
```



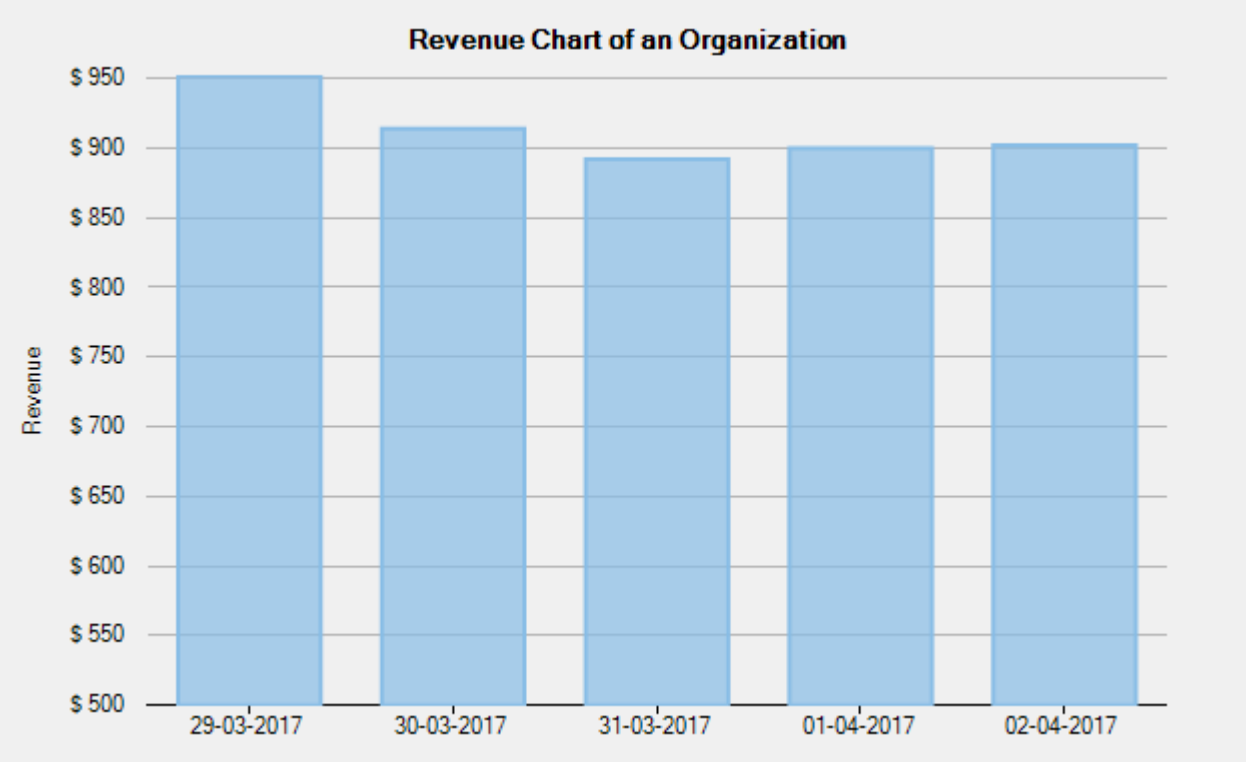
Axis Binding

Axis binding enables you to override the default axes labels that the axes display based on the chart binding. In other words, axis binding lets you show axes labels from a data source other than the chart data source.

FlexChart allows you to bind axes to a data source using the ItemsSource property of the Axis class. Specify the fields containing values for the axes labels in the data source using the Binding property of the Axis class.

The following image displays labels on Y-axis from the fields not part of the chart data source.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



The following code uses revenue data of an organization in a given year. The chart datasource contains the revenue data in the Euro currency. To replace euro currency axis labels with USD currency labels, the code binds Y-axis to a data source containing USD data.

- **Visual Basic**

```
' bind Y axis to a data source
flexChart.AxisY.ItemsSource = AxisData

' specify fields containing values for axis labels
flexChart.Binding = "Value,Text"
```

- **C#**

```
// bind Y axis to a data source
flexChart.AxisY.ItemsSource = AxisData;

// specify fields containing values for axis labels
flexChart.Binding = "Value,Text";
```

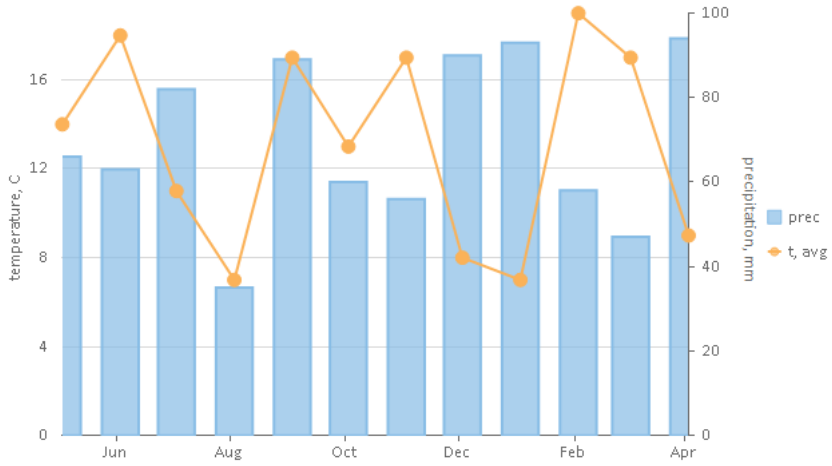
Multiple Axis

Although a chart contains primary X and Y axes, you may still sometimes require additional axes to fulfill your requirements. For example, you may want to plot series of a significantly different range of values in a chart. In addition, you may want to plot entirely different values (of different types) within a single chart. With just two axes, it would not be possible to display data in such scenarios effectively. In such cases, using secondary axes would come in handy. To use secondary axes, you can plot multiple series in a single chart with their own X and Y axes.

FlexChart allows you to work with multiple axes easily. You just need to create additional axes as per your requirements, and then bind the same to the AxisX and the AxisY property of a series.

The following image shows two Y axes. one primary and another auxiliary, along with X axis in FlexChart.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



The following code snippet demonstrates how you can create and use multiple axes in FlexChart:

- **XAML**

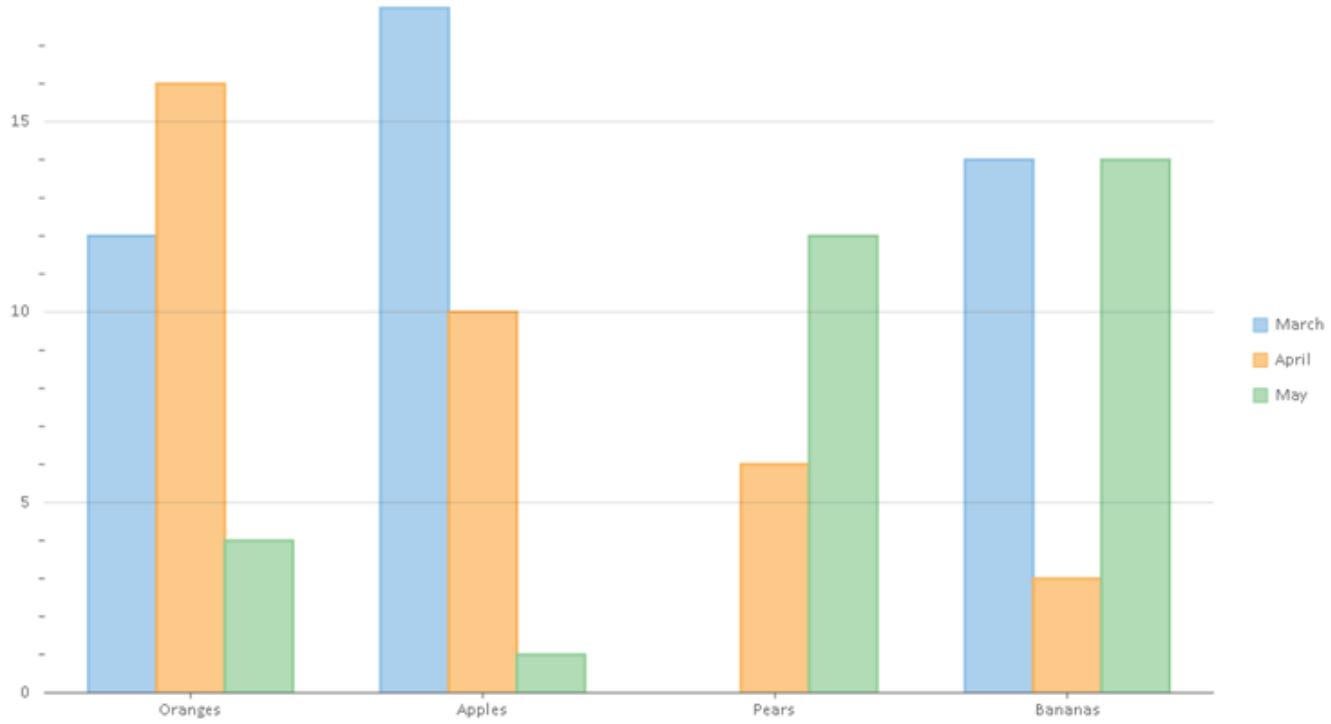
```
<c1:C1FlexChart x:Name="flexChart" ItemsSource="{Binding DataContext.Data}" BindingX="Time"
                    Grid.Row="1">
    <c1:Series x:Name="precSeries" SeriesName="prec" Binding="Precipitation">
        <c1:Series.AxisY>
            <c1:Axis Position="Right" Min="0" Max="100" Title="precipitation, mm" MajorGrid="False"/>
        </c1:Series.AxisY>
    </c1:Series>
    <c1:Series x:Name="avgSeries" SeriesName="t, avg" ChartType="LineSymbols" Binding="Temperature"/>
    <c1:C1FlexChart.AxisY>
        <c1:Axis Title="temperature, C" Min="0" MajorGrid="True" AxisLine="False" Position="Left" MajorTickMarks="None"></c1:Axis>
    </c1:C1FlexChart.AxisY>
</c1:C1FlexChart>
```

Axis Units

Axis units define at what intervals the tickmarks/gridlines should be displayed along the axis. FlexChart, by default, calculates the major and minor units automatically according to the data to be plotted on the chart. However, you can choose to change these intervals by using MajorUnit and MinorUnit property. This change impacts the values displayed on the value axis as well as the positioning of tick marks and grid lines if you choose to show them.

For example, in the FlexChart shown below, we have set the MajorUnit property to 5 and MinorUnit property to 1 on the Y-axis. You will notice that the difference between each major tick mark is 5 and minor tick mark is 1.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



Use the following code to set the MajorUnit and MinorUnit properties.

In XAML

```
<c1:C1FlexChart.AxisY>
    <c1:Axis Format="0.00" MajorUnit="5" MinorUnit="1"></c1:Axis>
</c1:C1FlexChart.AxisY>
```

In Code

```
flexChart.AxisY.MajorUnit = 5;
flexChart.AxisY.MinorUnit = 1;
```

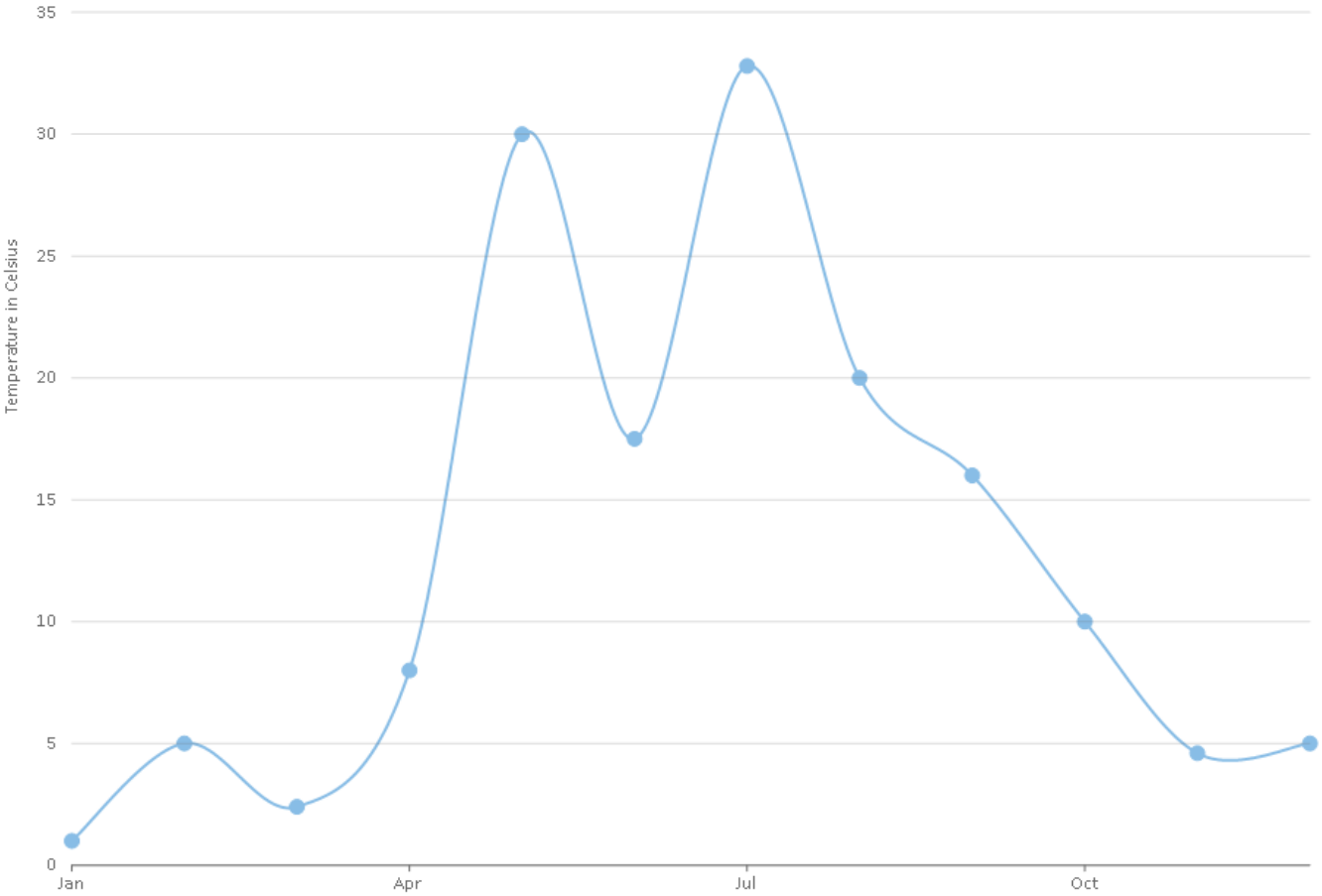
DateTime Axis Units

In case of DateTime axis, FlexChart provides you an option to set the time unit as well using the TimeUnit property. You can choose from day, month, quarter, week, and year options. This property accepts value from the TimeUnits enumeration. The enumeration includes the following values.

- Day
- Week
- Month
- Quarter
- Year

For example, in order to set major unit of a value axis to 3 months, you first need to set the **TimeUnit** property to Month, and then set the **MajorUnit** property to 3.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



Use the following code to set the **TimeUnit** and **MajorUnit** properties.

In XAML

```
<c1:C1FlexChart.AxisX>
   <c1:Axis Format="0.00" TimeUnit="Month" MajorUnit="3"></c1:Axis>
</c1:C1FlexChart.AxisX>
```

In Code

```
flexChart1.AxisX.TimeUnit = C1.Chart.TimeUnits.Month;
flexChart1.AxisX.MajorUnit = 3;
```

FlexChart Axes Labels

Axes labels are the values that appear along the axes. By default, axes labels are determined on the basis of the axes data points and the generated intervals.

In **FlexChart**, you can change the look, format, and alignment of the axes labels using the properties listed below.

Property	Description
Format	Specifies the format string used for the axes labels.
LabelAlignment	Sets the alignment of the axes labels.
LabelAngle	Specifies the rotation angle of the labels.

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Labels	Indicates whether the axes labels are visible.
OverlappingLabels	Indicates how to handle overlapping labels.

The following sections discuss how to work with these properties:

- Axes Labels Format
- Axes Labels Rotation
- Axes Labels Visibility
- Axes Labels Overlap
- Axis Grouping

Axes Labels Format

By default, axis labels are determined automatically based on the data points and generated intervals of the axes. However, you can still format the axis labels by using the Format property to cover your requirements better.

The **Format** property accepts values from the Standard .Net Format string.

Axes Labels Rotation

When the horizontal axis is crowded with axis labels, you need to rotate the labels to avoid the cluttered look. Rotating the labels allows you to accommodate a large number of labels in a limited space on the axis.

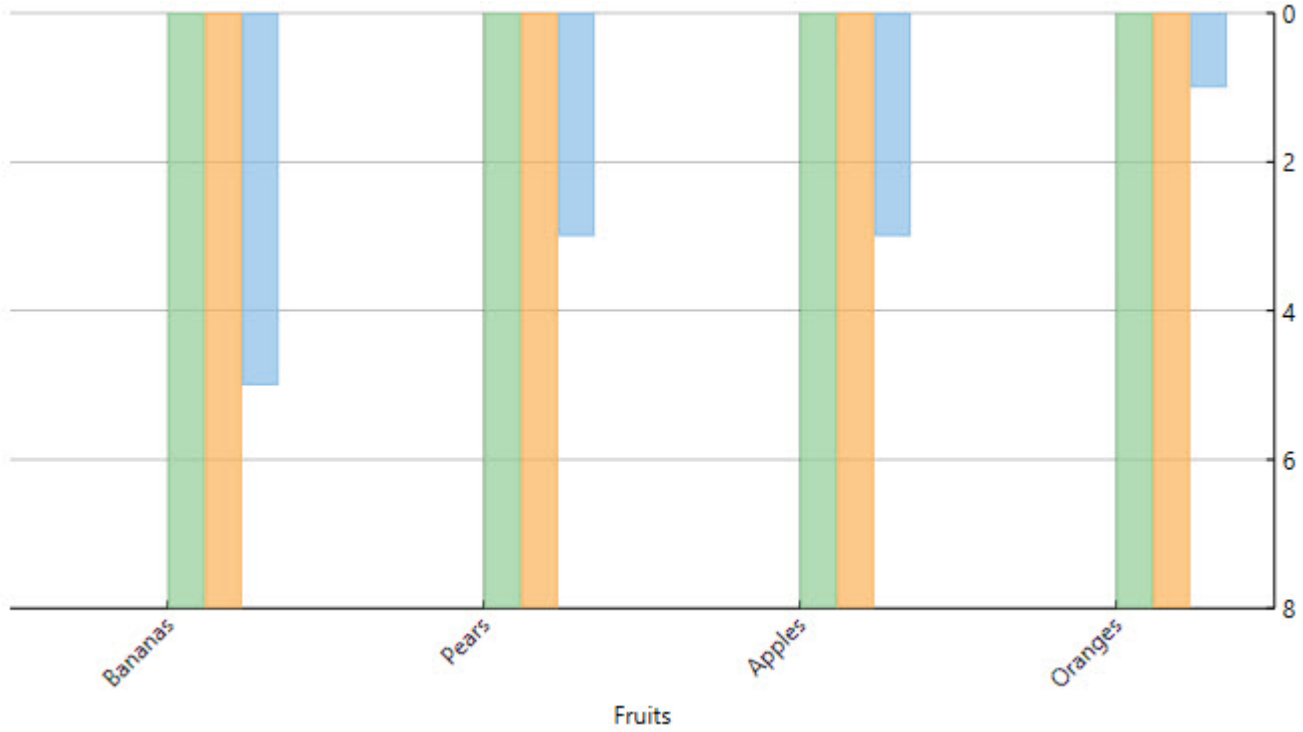
You can use the LabelAngle property to rotate axes labels in anticlockwise direction in **FlexChart**.

See the code given below for reference.

- **C#**

```
flexChart.AxisX.LabelAngle = 45;
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.



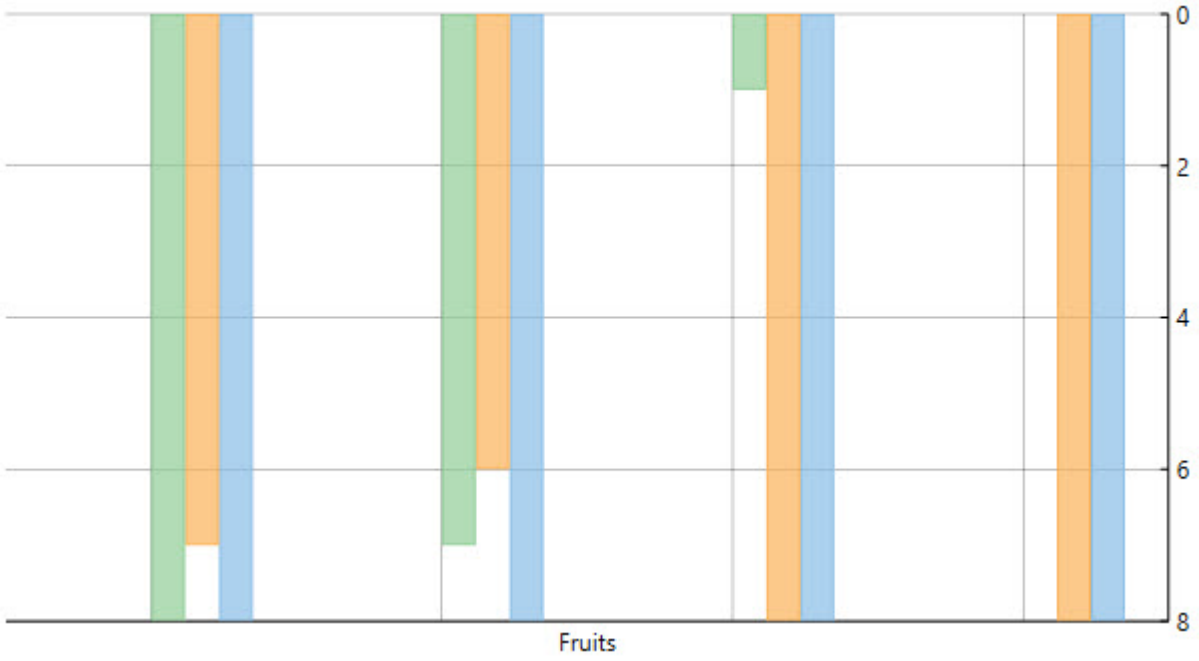
Axes Labels Visibility

FlexChart enables you to show or hide axis labels with the Labels property. You can set the property to False for a specific axis, if you want to hide axis labels along the axis. The default value of the **Labels** property is True.

See the following code snippet:

- **C#**

```
flexChart.AxisX.Labels = false;
```



Copyright © 2019 GrapeCity, Inc. All rights reserved.

Axes Labels Overlap

In case there are less number of data points and shorter label text, axis labels are rendered without any overlapping. However, axis labels may overlap due to its long text or large numbers of data points in chart.

To manage overlapped axis labels in FlexChart, use the following options.

- **Trim or Wrap Axis Labels**
- **Staggered Axis Labels**

Trim or Wrap Axis Labels

In case there are overlapping labels in the chart for any reason, you can manage the same using the OverlappingLabels property.

The OverlappingLabels property accepts the following values in the OverlappingLabels enumeration:

Property	Description
Auto	Hides overlapping labels.
Show	Shows all labels including the overlapping ones.
Trim	Trim label, if it's larger than the available width.
WordWrap	Wrap label, if it's larger than the available width.

Here is the code snippet:

- **C#**

```
flexChart.AxisX.OverlappingLabels = C1.Chart.OverlappingLabels.Auto;
flexChart.AxisY.OverlappingLabels = C1.Chart.OverlappingLabels.Show;
```

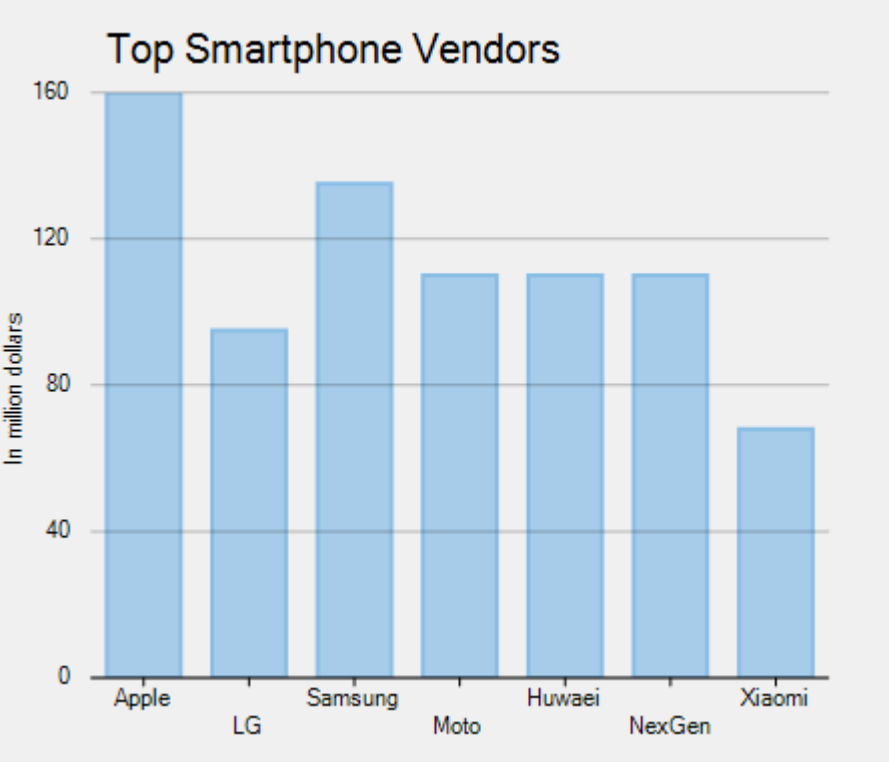
Staggered Axis Lines

Another way to handle overlapping of axis labels is to stagger them for better visibility. Staggered axis labels can generated by using StaggeredLines property. This property accepts an integer value and the default value is set to 1.

C#

```
// Set StaggeredLines property
flexChart1.AxisX.StaggeredLines = 2;
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.



Axis Grouping

FlexChart provides the flexibility to group axis labels as per the requirement. Axis grouping helps in improving the readability of the chart and makes it easy for analyzing data from different levels. Implementation of axis grouping in FlexChart depends on the data you are using, it can be either categorical data, numerical data, or DateTime data.

FlexChart supports the following axis grouping depending upon the data.

Categorical Axis Grouping
: Learn how to perform axis grouping while working with categorical data.

Numerical Axis Grouping
: Learn how to perform axis grouping while working with numerical data.

DateTime Axis Grouping
: Learn how to perform axis grouping while working with date time format data.

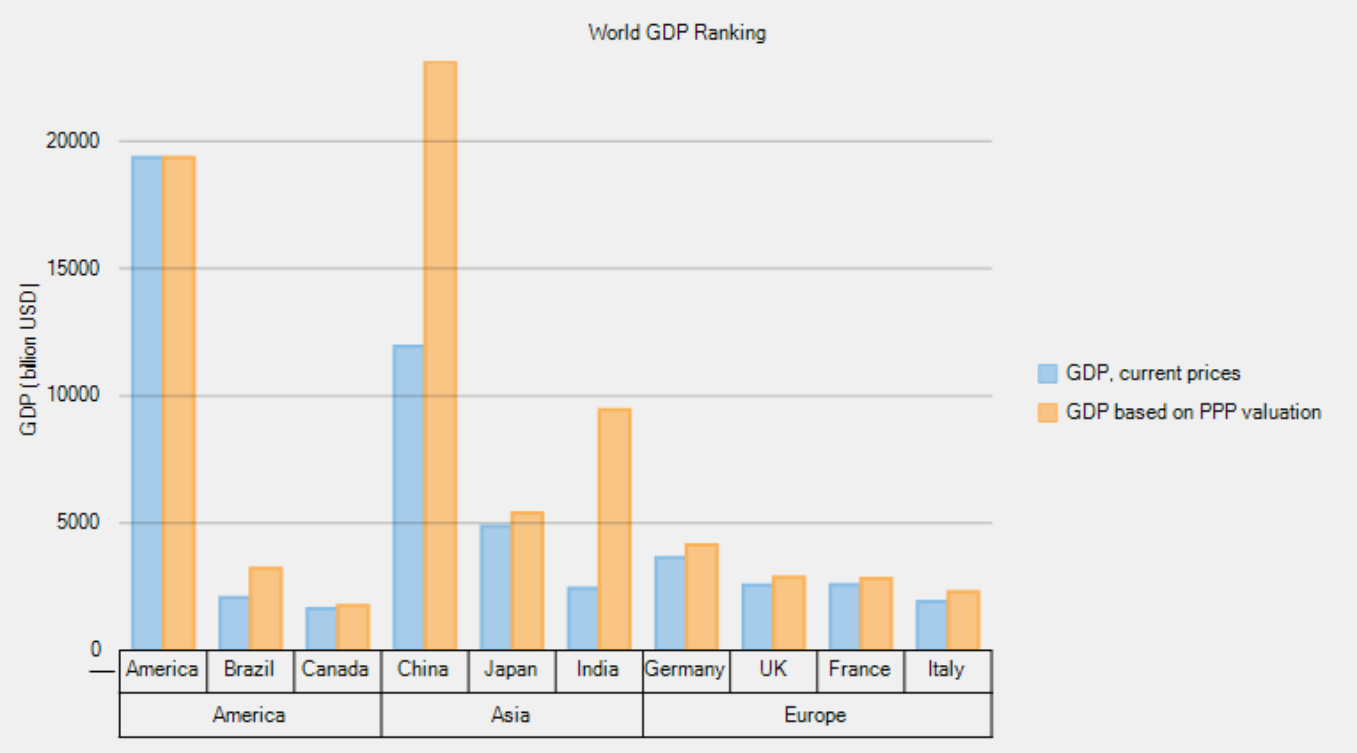
Categorical Grouping

Categorical axis grouping is applicable in scenarios where the data displayed on the axis is categorical in nature. Categorical data can either be flat or hierarchical. In case you are using flat data, use the GroupNames property to apply axis grouping. And, in case you are using hierarchical data, use the GroupNames and GroupItemsPath property to apply axis grouping.

Moreover, FlexChart allows you to set the group separator using the GroupSeparator property. Also, it allows you to expand or collapse group levels by setting the GroupVisibilityLevel property.

The following image shows how FlexChart appears after setting the categorical axis grouping using flat data.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



Add the following code in Index.xaml.

XAML

```
<c1:C1FlexChart x:Name="flexChart" Background="White" ChartType="Column"
BindingX="Country"
ItemsSource="{Binding Data}" ToolTipContent="{}{x}
{seriesName}{y:n0}" Header="World GDP Ranking" Grid.Row="1" >
            <c1:C1FlexChart.HeaderStyle>
                <c1:ChartStyle FontSize="20" FontFamily="GenericSansSerif"/>
            </c1:C1FlexChart.HeaderStyle>
            <c1:Series SeriesName="GDP, current prices" Binding="CurrentPrices"/>
            <c1:Series SeriesName="GDP based on PPP valuation"
Binding="PPPValuation"/>
            <c1:C1FlexChart.AxisX>
                <c1:Axis GroupSeparator="Gird" GroupNames="Continent"
GroupVisibilityLevel="-2" />
            </c1:C1FlexChart.AxisX>
            <c1:C1FlexChart.AxisY>
                <c1:Axis Title="GDP (billion USD)" MajorGrid="True"/>
            </c1:C1FlexChart.AxisY>
        </c1:C1FlexChart>
```

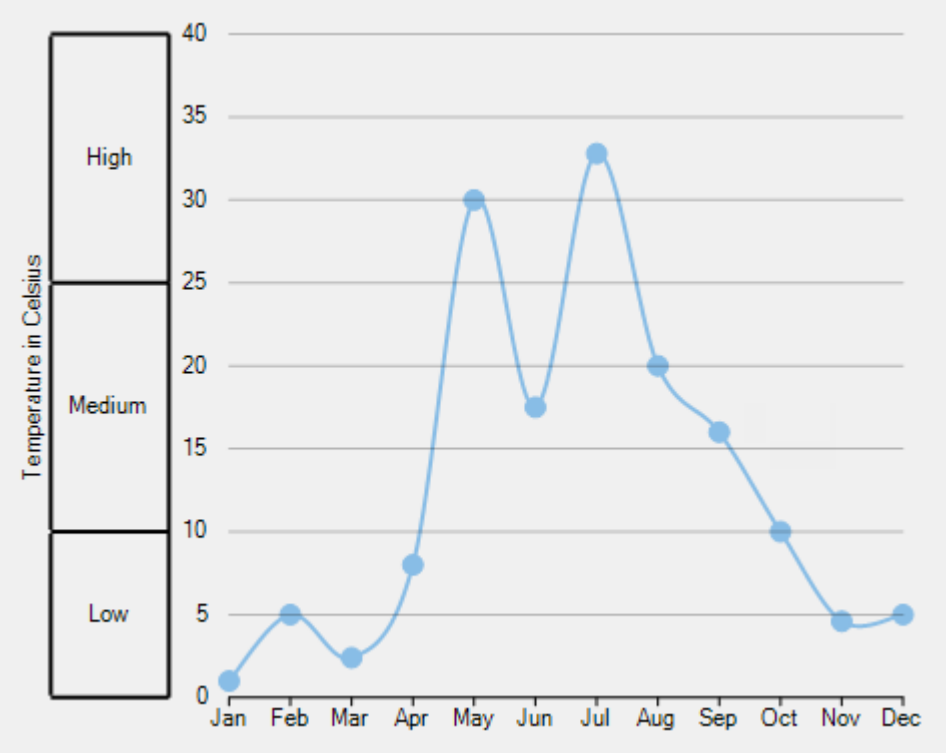
Numerical Axis Grouping

Numerical axis grouping is applicable in scenarios where the data displayed on the axis represents numeric values. To implement numerical axis grouping in FlexChart, set the GroupProvider property to an object of the IAxisGroupProvider implementation.

Copyright © 2019 GrapeCity, Inc. All rights reserved.

In the example code below, we have created a class **NumericAxisGroupProvider** that implements the IAxisGroupProvider interface. The interface provides GetLevels method that returns the group levels and GetRanges method that returns the group ranges for a given level. Moreover, FlexChart allows you to set the group separator using the GroupSeparator property.

The following image shows how FlexChart appears after setting the numerical axis grouping.



Add the following code in Index.xaml.

XAML

```
<c1:C1FlexChart x:Name="flexChart" Background="White" ChartType="SplineSymbols"
BindingX="Month"
      ItemsSource="{Binding Data}" Grid.Row="1" >
            <c1:Series Binding="Temperature" />
            <c1:C1FlexChart.AxisY>
            <c1:Axis Title="Temperature in Celsius" MajorGrid="True"
GroupSeparator="Horizontal" Min="0" Max="40"/>
            </c1:C1FlexChart.AxisY>
</c1:C1FlexChart>
```

Code

```
public NumericAxisGrouping()
        {
            InitializeComponent();
            flexChart.AxisY.GroupProvider = new NumericAxisGroupProvider();
        }
        class NumericAxisGroupProvider : IAxisGroupProvider
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
{
    public int GetLevels(IRange range)
    {
        return 1;
    }

    public IList<IRange> GetRanges(IRange range, int level)
    {
        var ranges = new List<IRange>();
        if (level == 1)
        {
            ranges.Add(new DoubleRange("Low", 0, 10));
            ranges.Add(new DoubleRange("Medium", 10, 25));
            ranges.Add(new DoubleRange("High", 25, 40));
        }
        return ranges;
    }
}
```

DateTime Axis Grouping

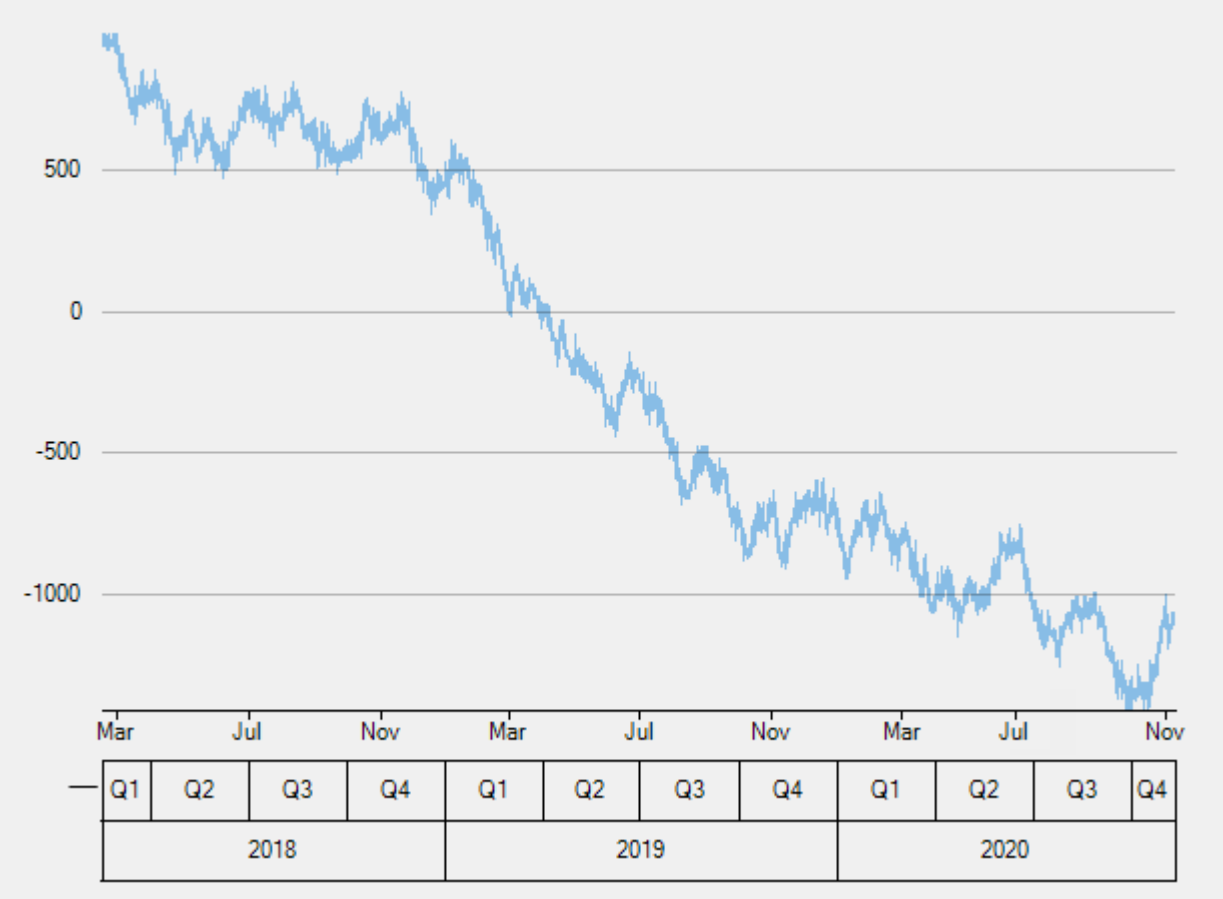
DateTime axis grouping is applicable in scenarios where the data displayed on the axis represents date time values. To implement date axis grouping in FlexChart, set the GroupProvider property to an object of the IAxisGroupProvider implementation.

In the example code below, we have created a class **DateTimeGroupProvider** that implements the IAxisGroupProvider interface. The interface provides GetLevels method that returns the group levels and GetRanges method that returns the group ranges for a given level.

Moreover, FlexChart allows you to set the group separator using the GroupSeparator property. Also, it allows you to expand or collapse group levels by setting the GroupVisibilityLevel property.

The following image shows how FlexChart appears after setting the date axis grouping.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



Add the following code in Index.xaml.

XAML

```
<c1:C1FlexChart x:Name="flexChart" Background="White" ChartType="Line"
BindingX="Time"
            ItemsSource="{Binding Data}" Grid.Row="1" >
            <c1:Series Binding="Price"/>
            <c1:C1FlexChart.AxisX>
                <c1:Axis GroupSeparator="Grid" GroupVisibilityLevel="1"
Format="MMM"/>
            </c1:C1FlexChart.AxisX>
        </c1:C1FlexChart>
```

Code

```
public DateTimeAxisGrouping()
        {
            InitializeComponent();
            flexChart.AxisX.GroupProvider = new DateTimeGroupProvider();
        }
        public class DateTimeGroupProvider : IAxisGroupProvider
        {
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
            public int GetLevels(IRange range)
            {
                return 2;
            }

            public IList<IRange> GetRanges(IRange range, int level)
            {
                var timeRange = range as TimeRange;
                if (timeRange == null)
                    return null;
                var min = timeRange.TimeMin;
                var max = timeRange.TimeMax;
                var span = max - min;

                List<IRange> ranges = new List<IRange>();
                DateTime start;
                if (level == 1)
                {
                    start = new DateTime(min.Year,
((int)Math.Ceiling((double)min.Month / 3) - 1) * 3 + 1, 1);
                    ranges = Enumerable.Range(0, ((max.Month - start.Month) / 3 + 1)
+ 4 * (max.Year - start.Year)).Select(a => start.AddMonths(a * 3))
                   .TakeWhile(a => a <= max)
                   .Select(a => (IRange)(new TimeRange("Q" +
(int)Math.Ceiling((double)a.Month / 3), a, a.AddMonths(3)))).ToList();
                }
                else
                {
                    start = new DateTime(min.Year, 1, 1);
                    ranges = Enumerable.Range(0, max.Year - start.Year + 1).Select(a
=> start.AddYears(a))
                   .TakeWhile(a => a <= max)
                   .Select(a => (IRange)(new TimeRange(a.ToString("yyyy"), a,
a.AddYears(1)))).ToList();
                }

                return ranges;
            }
        }
```

Annotations

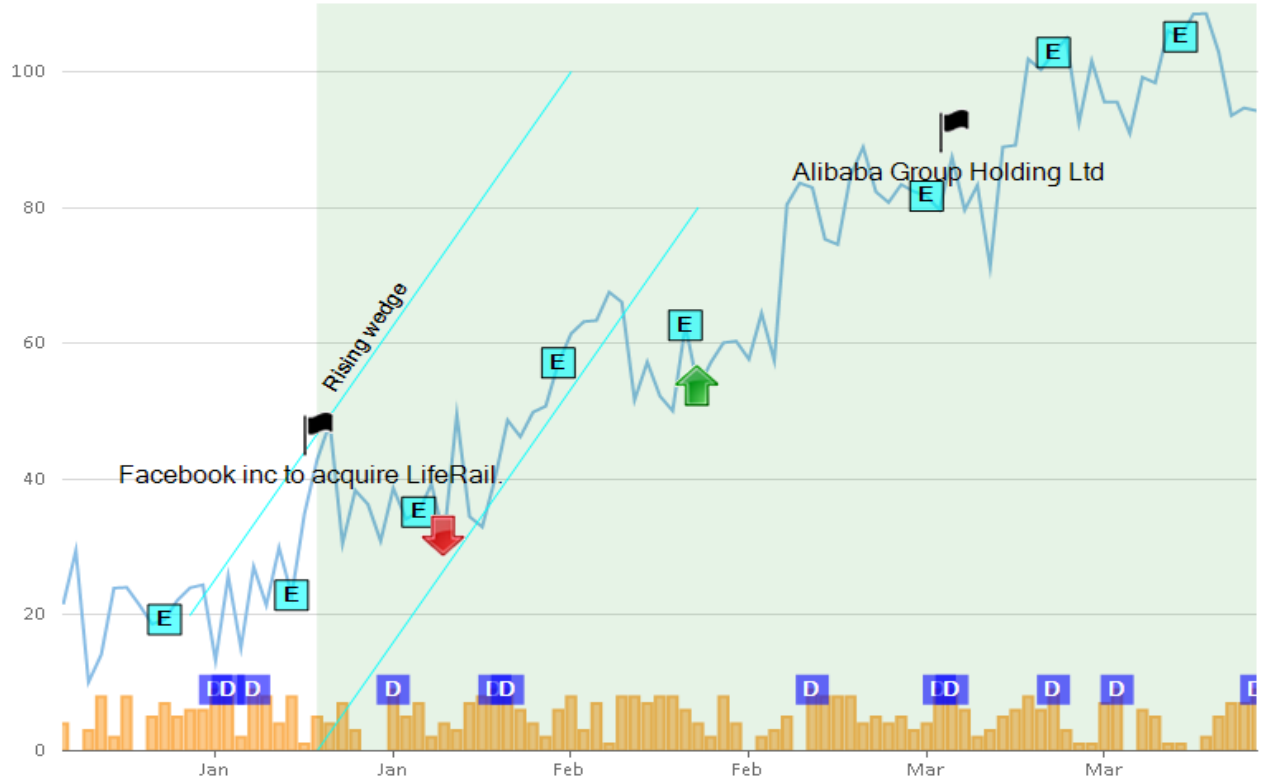
Annotations are visual elements used to mark or highlight specific areas in a chart. They include texts, images, and shapes that can be used to display and highlight important information about specific data points. The primary purpose of using annotations in a chart is to communicate the chart data clearly.

FlexChart provides eight types of annotations in three categories: shape, text, and image annotations. Each annotation type allows you to make chart data informative in different ways. Those include displaying information in circle, rectangle, polygon, and other shapes, and highlighting data through explanatory notes or images.

In addition, annotations in FlexChart can be positioned in the chart using attachment modes, such as Absolute, Relative, Data Index, and Data Coordinate. Both annotations and their content are customizable through styling

Copyright © 2019 GrapeCity, Inc. All rights reserved.

properties of font, color, and stroke. They can be made interactive by adding tooltips, especially image annotations.



To explore annotations, refer to the following sections:

- Adding Annotations
- Positioning Annotations
- Customizing Annotations
- Types of Annotations
- Creating Callouts

Adding Annotations

FlexChart enables you to add annotations in an annotation layer, which contains the collection of all annotations in the chart.

To add annotations in FlexChart, follow these steps:

1. Create an annotation layer in FlexChart.
2. Add the annotation instance in the annotation layer.

To create an annotation layer in FlexChart, create an instance of the AnnotationLayer class and add it to the Layers collection of FlexChart. To add an annotation into the annotation layer, create an instance of the annotation class based on its type. Add the annotation instance to the Annotations collection of the annotation layer.

The following code snippet illustrates how to create and add the Rectangle annotation to Annotation Layer in FlexChart.

XAML

```
<c1:C1FlexChart.Layers>
    <c1:AnnotationLayer>
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
        <c1:AnnotationLayer.Annotations>
            <c1:Rectangle Content="Maximum Tax Revenue&#13;2013&#13;45000">
            </c1:Rectangle>
        </c1:AnnotationLayer.Annotations>
    </c1:AnnotationLayer>
</c1:C1FlexChart.Layers>
```

Positioning Annotations

In FlexChart, positioning annotations includes two mechanisms (not necessarily in the same order), as follows:

- Positioning annotations relative to the chart.
- Positioning annotations relative to the data points.

Positioning Annotations Relative to the Chart

Positioning annotations relative to the chart includes specifying the attachment and the location of the annotations in the chart.

FlexChart provides four ways of attaching annotations, as follows:

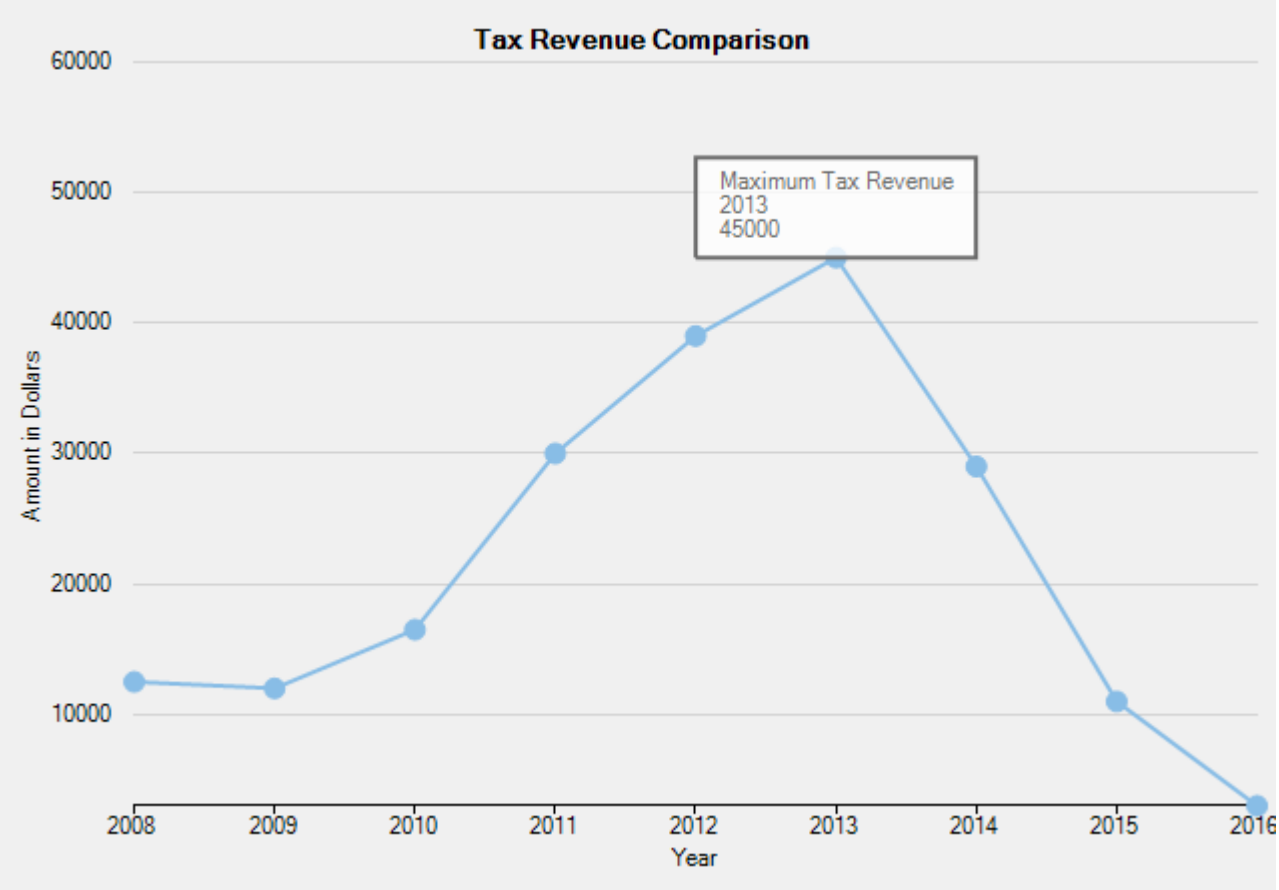
- **Absolute**: This attachment indicates that the annotation is fixed and cannot move, irrespective of the resizing of the application. To set the absolute attachment, set the Attachment property to **Absolute** from the AnnotationAttachment enum. To set the location of the annotation in the absolute attachment mode, set the annotation's coordinates in pixels.
- **DataCoordinate**: This attachment indicates that the annotation is attached to a specific data point. To set this attachment, set the Attachment property to **DataCoordinate** from the AnnotationAttachment enum. To set the annotation's location, specify the annotation's data coordinates by setting the Location property.
- **DataIndex**: This attachment indicates that the annotation is attached to the series as per the series index and to the point as per the point index. To set this attachment, set the Attachment property to **DataIndex** from the AnnotationAttachment enum. To specify the annotation's location, set the SeriesIndex and the PointIndex properties.
- **Relative**: This attachment indicates that the annotation retains its location and dimensions relative to the chart. To set this attachment, set the Attachment property to **Relative** from the AnnotationAttachment enum. Specify the annotation's location using the Location property in terms of relative position inside the chart where (0, 0) is the top left corner and (1, 1) is the bottom right corner.

Positioning Annotations Relative to the Data Points

Specify the position of annotations with respect to the data points by setting the Position property from the AnnotationPosition enum.

The following image displays the Rectangle annotation highlighting the maximum tax revenue in the year, 2013.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



The following code compares tax revenue data of nine consecutive years to display the maximum tax revenue. The code shows how to specify the attachment, location, and position of the Rectangle annotation to Annotation Layer in FlexChart.

XAML

```
<c1:Rectangle Content="Maximum Tax Revenue&#13;2013&#13;45000"
              Attachment="DataIndex"
              SeriesIndex="0"
              PointIndex="5"
              Position="Top"
              Width="140"
              Height="50">
</c1:Rectangle>
```

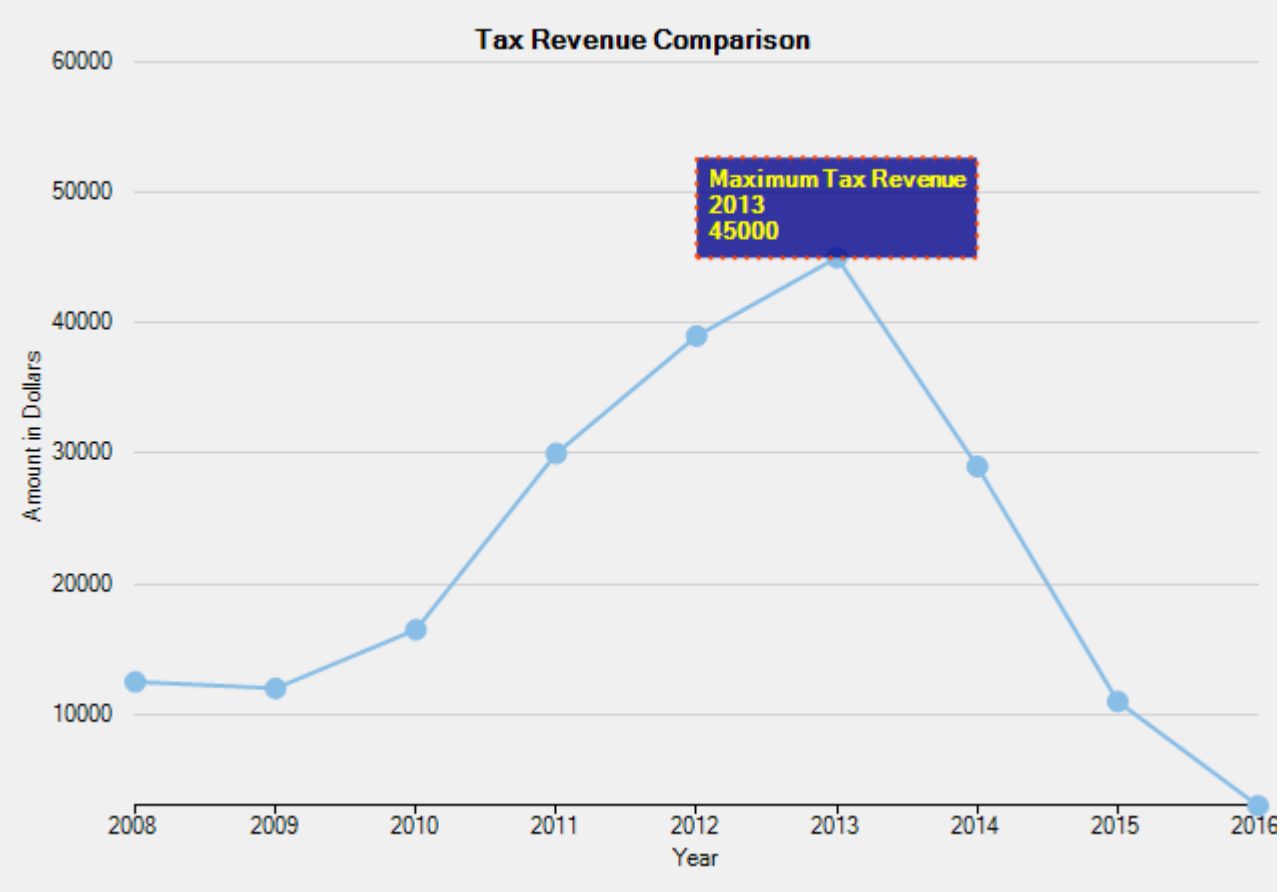
Customizing Annotations

FlexChart annotations are customizable in terms of dimensions (for shapes), scaling (for images), and content style (for all except images).

- **Dimensions**: Change the dimensions of all shapes using dimension properties of the respective classes. For instance, to change the dimensions of the Rectangle annotation, set the Height and the Width properties of the Rectangle class.
- **Style**: Customize the appearance of shape and text annotations in terms of color, font, and stroke by using the Style property of the AnnotationBase class.
- **Content Style**: Customize the appearance of content present in shape annotations by using the ContentStyle property of the Shape class.

Copyright © 2019 GrapeCity, Inc. All rights reserved.

The following image displays the Rectangle annotation customized to further highlight the maximum tax revenue in the year, 2013.



The following code compares tax revenue data of nine consecutive years to display the maximum tax revenue. The code shows how to set the Rectangle annotation's dimensions, customize its appearance and content.

XAML

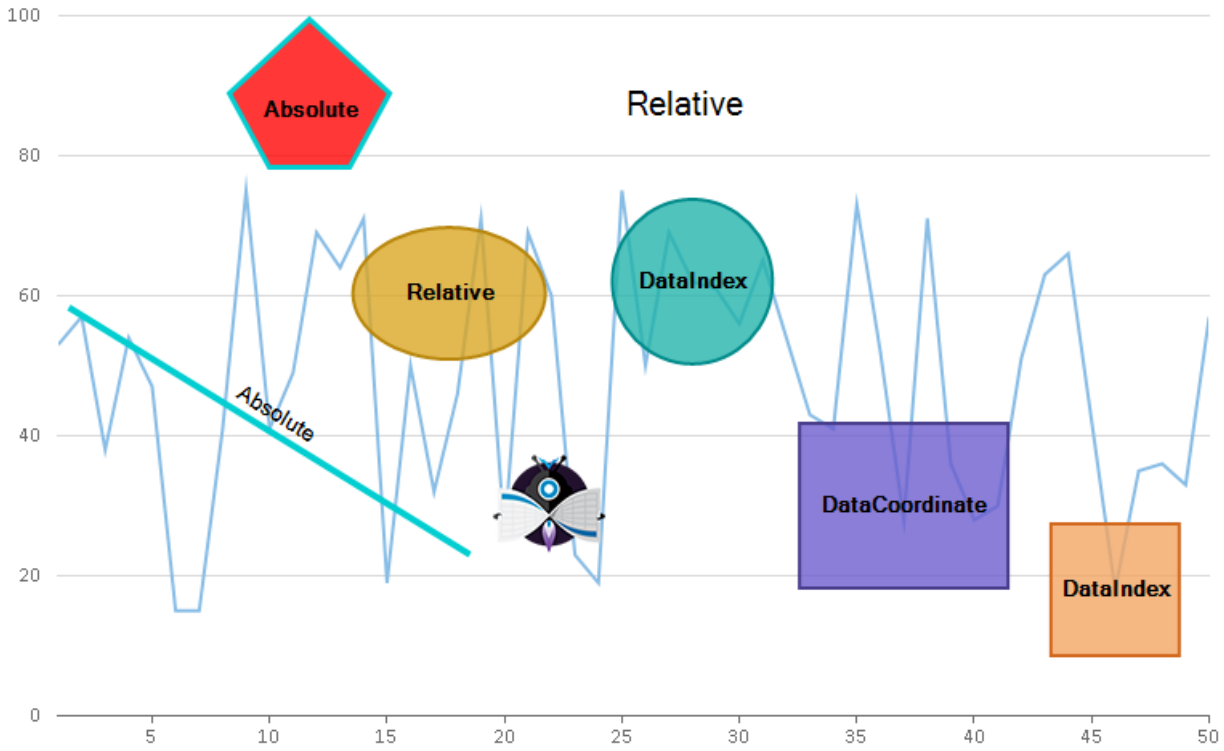
```
<c1:Rectangle.Style>
    <c1:ChartStyle Fill="DarkBlue"
                   Stroke="OrangeRed"
                   StrokeThickness="2"
                   StrokeDashArray="1,2"
                   FontFamily="GenericSansSerif"
                   FontWeight="Regular" />
</c1:Rectangle.Style>
<c1:Rectangle.ContentStyle>
    <c1:ChartStyle Stroke="Yellow"
                   FontFamily="GenericSansSerif"
                   FontSize="8.5"
                   FontWeight="Bold">
    </c1:ChartStyle>
</c1:Rectangle.ContentStyle>
```

Types of Annotations

FlexChart provides eight types of annotations in three categories, as follows:

Copyright © 2019 GrapeCity, Inc. All rights reserved.

- **Shapes**: Include useful information at specific areas and highlight the areas within chart data by using shapes, such as Circle, Ellipsis, Rectangle, Square, Line, or Polygon.
- **Text**: Add descriptive notes or informative comments at specific points in the chart by using text annotations.
- **Image**: Add self-explanatory images to readily communicate chart data by using image annotations.



To explore different types of annotations provided by FlexChart, refer to the following sections:

- Shape Annotations
- Text Annotations
- Image Annotations

Shape Annotations

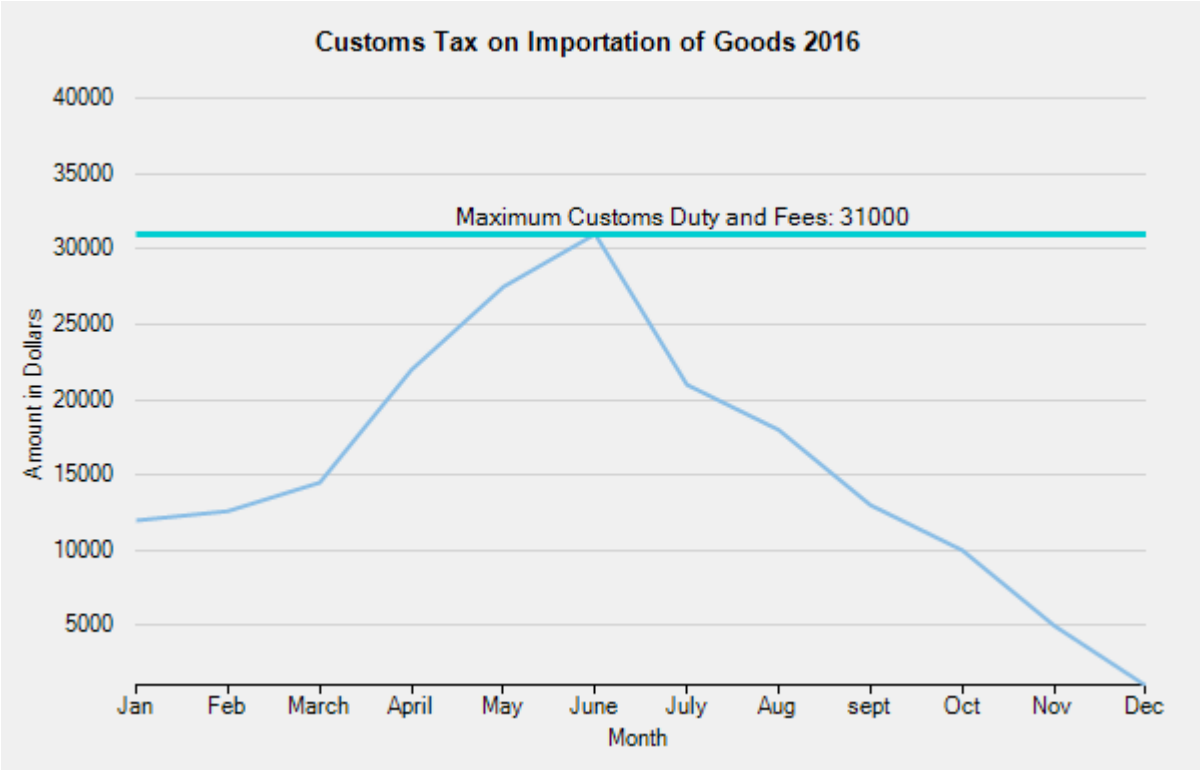
Shapes are beneficial for drawing the user's attention at specific areas where important data is highlighted.

FlexChart offers six shape annotations, as follows:

- Circle
- Ellipsis
- Line
- Polygon
- Rectangle
- Square

The following image shows the Line annotation highlighting the maximum customs duty and fees in the year, 2016.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



To create any of these specific shapes, create an instance of the shape annotation's class. Set the dimensions of the shape by using the dimension properties from the corresponding class. For example, to create a line annotation, create an instance of the Line class. Specify the length of the line annotation or rotate it by setting the Start and the End properties of the Line class.

For any shape annotation, specify the text by setting the Content property of the Shape class, the base class for all shape annotations. In addition, other shapes like triangles and arrows can be created using the Polygon annotation in FlexChart.

The following code uses customs tax data on importation of goods for representing its increment or decrement for the year, 2016. The code shows how to add, position, and customize the Line annotation in FlexChart.

XAML

```
<c1:C1FlexChart.Layers>
    <c1:AnnotationLayer>
        <c1:AnnotationLayer.Annotations>
            <c1:Line Content="Maximum Customs Duty and Fees: 31000"
                     Attachment="DataCoordinate"
                     Start="0,31000"
                     End="12,31000"
                     Position="Top">
                <c1:Line.Style>
                    <c1:ChartStyle Stroke="DarkTurquoise"
                                   StrokeThickness="3"/>
                </c1:Line.Style>
                <c1:Line.ContentStyle>
                    <c1:ChartStyle Stroke="Black"
                                   FontFamily="GenericSansSerif"
                                   FontSize="9"
                                   FontWeight="Bold">
                    </c1:ChartStyle>
                </c1:Line.ContentStyle>
            </c1:Line>
```

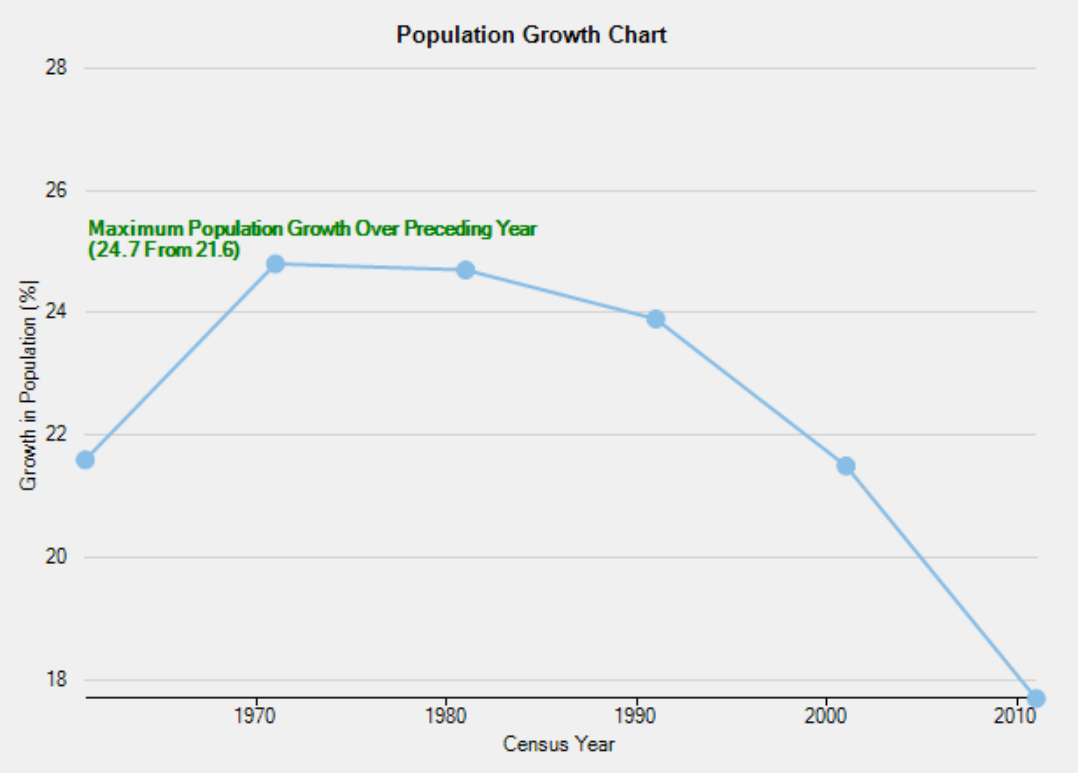
Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
        </c1:AnnotationLayer.Annotations>
    </c1:AnnotationLayer>
</c1:C1FlexChart.Layers>
```

Text Annotation

Text annotations let you add additional information at specific data points to make the data informative. FlexChart lets you add single line as well as multiline text in text annotations.

The following image shows the Text annotation displaying the maximum population growth rate between 1961 and 2011.



To work with text annotation in FlexChart, create an instance of the Text class and set the Content property for the instance.

The following code compares population growth rates at specific years in five consecutive decades. The code shows how to add, position, and customize the Text annotation in FlexChart.

XAML

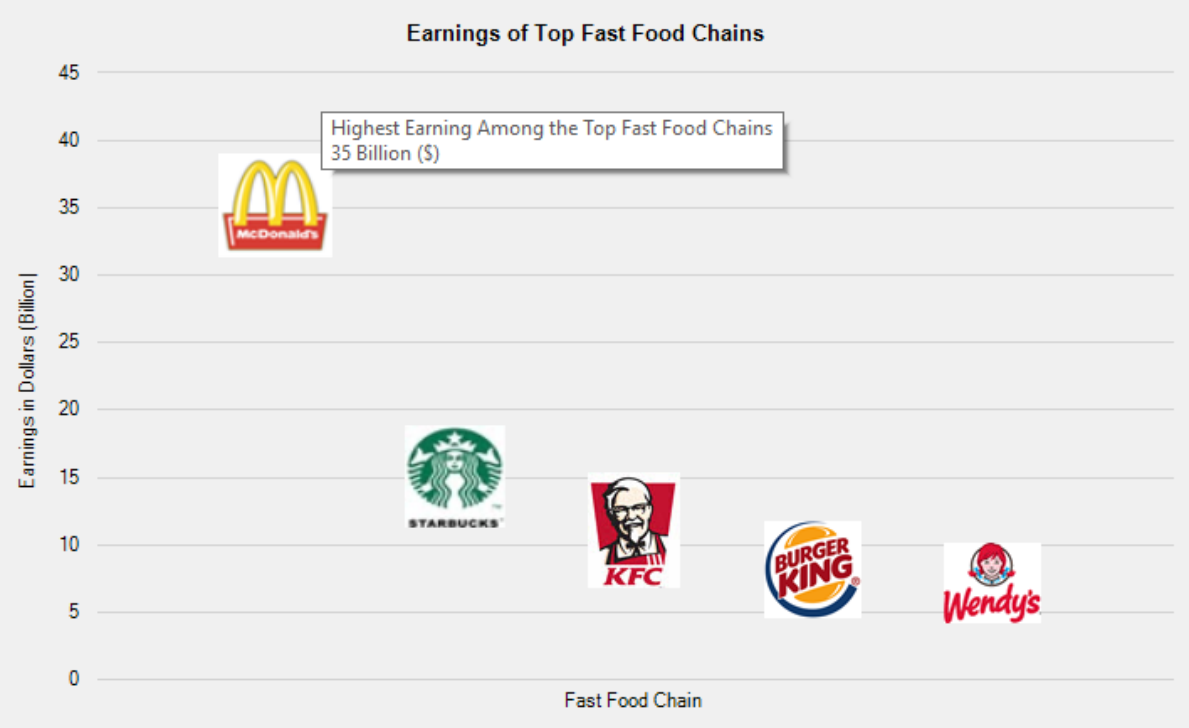
```
<c1:C1FlexChart.Layers>
    <c1:AnnotationLayer>
        <c1:AnnotationLayer.Annotations>
            <c1:Text Content="Maximum Population Growth Over Preceding Year&#x0a;(24.7 From 21.6)"
                     Attachment="DataCoordinate"
                     Location="1961,25.15"
                     Position="Top">
                <c1:Text.Style>
                    <c1:ChartStyle Stroke="Green"
                                   FontFamily="GenericSansSerif"
                                   FontSize="8"
                                   FontWeight="Bold" />
                </c1:Text.Style>
            </c1:Text>
        </c1:AnnotationLayer.Annotations>
    </c1:AnnotationLayer>
</c1:C1FlexChart.Layers>
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Image Annotation

Image annotations create visual impact and allow users to quickly interpret the chart data. A great way to communicate informative data through image annotations is by adding tooltips.

The following image displays highest earning of a fast food chain among others using a tooltip with an image annotation.



To use image annotations in FlexChart, create an instance of the Image class and set an image for the instance by specifying the image path in the Source property. Scale the image or adjust its size by setting the Height and the Width properties. To add tooltips to image annotations, set the TooltipText property of the AnnotationBase class for the image annotation instance.

The following code compares earnings of the top fast food chains in United States. The code shows how to add, position, and customize the Image annotation in FlexChart.

XAML

```
<c1:C1FlexChart.Layers>
    <c1:AnnotationLayer>
        <c1:AnnotationLayer.Annotations>
            <c1:Image Source="C:\\Resources\\image1.png"
                Attachment="DataCoordinate"
                      Location="1,35"
                      Position="Center"
                      Width="68"
                      Height="62"
                      TooltipText="Highest Earning Among the Top Fast Food Chains\n35 Billion ($)">
            </c1:Image>
            <c1:Image Source="C:\\Resources\\image2.png"
                Attachment="DataCoordinate"
                      Location="2,15"
                      Position="Center"
                      Width="60"
                      Height="61">
            </c1:Image>
            <c1:Image Source="C:\\Resources\\image3.png"
                Attachment="DataCoordinate"
                      Location="3,11"
                      Position="Center">
            </c1:Image>
            <c1:Image Source="C:\\Resources\\image4.png"
                Attachment="DataCoordinate"
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
                    Location="4,8"
                    Position="Center">
            </c1:Image>
            <c1:Image Source="C:\\Resources\\image5.png"
                Attachment="DataCoordinate"
                    Location="5,7"
                    Position="Center">
            </c1:Image>
        </c1:AnnotationLayer.Annotations>
    </c1:AnnotationLayer>
</c1:C1FlexChart.Layers>
```

Creating Callouts

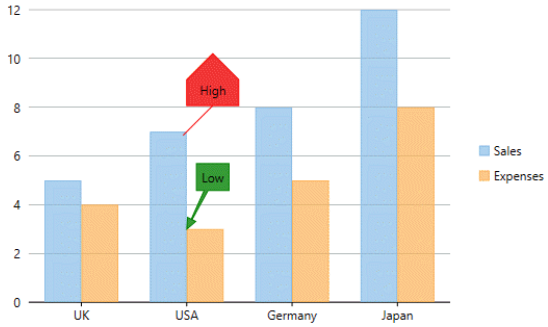
Callouts in charts are used to display the details of a data series or individual data points in an easy-to-read format. Callouts being connected with data points, help better visualize and comprehend chart data by minimizing visual disturbances in the chart area. In FlexChart, Polygon type annotations can be customized to create chart callouts with line or arrow connectors.

In this example, we are using sample created in the Quick Start topic to further create an arrow callout and polygon annotation with line connection. This is done with the help of the Points property and the ContentCenter property that define the coordinates of polygon vertices and annotation content center respectively.

To create callouts connected with respective data points, follow these steps:

- **Step 1: Create annotation with line connector**
- **Step 2: Create arrow annotation callout**
- **Step 3: Render the annotations in chart**

The following image illustrates polygon annotations connected to data points through arrow and line connectors.



Step 1: Create annotation with line connector

To create a line callout, use the following code.

- **Visual Basic**

```
        ...
        ' Create and stylize a line callout annotation of polygon type
        Dim lineCallout = New C1.WPF.Chart.Annotation.Polygon() With {
    .Content = "High",
    .Style = New ChartStyle() With {
        .Fill = New SolidColorBrush(Colors.Red) With {
            .Opacity = 200.0 / 255
        },
        .Stroke = New SolidColorBrush(Colors.Red)
    },
    .Attachment = AnnotationAttachment.DataIndex,
    .SeriesIndex = 0,
    .PointIndex = 1,
    .ContentCenter = New Point(25, -40),
    .Points = New PointCollection(New Point() {New Point(0, 0), New Point(25, -25), New Point(50, -25), New Point(50, -50), New Point(25, -75), New Point(0, -50),
        New Point(0, -25), New Point(25, -25), New Point(0, 0)})
}
        ...
```

- **C#**

```
    ...
    // Create a line callout annotation of polygon type
    var lineCallout = new C1.WPF.Chart.Annotation.Polygon()
    {
        Content = "High",
        //Stylise the line callout annotation of polygon type
        Style = new ChartStyle()
        {
            Fill = new SolidColorBrush(Colors.Red) { Opacity = 200.0 / 255 },
            Stroke = new SolidColorBrush(Colors.Red),
        },
        Attachment = AnnotationAttachment.DataIndex,
        SeriesIndex = 0,
        PointIndex = 1,
        ContentCenter = new Point(25, -40),
        Points = new PointCollection(new Point[] { new Point(0, 0), new Point(25, -25),
new Point(50, -25), new Point(50, -50), new Point(25, -75),
new Point(0, -50), new Point(0, -25), new Point(25, -25), new Point(0, 0) })
    };
    ...
```

Back to Top

Step 2: Create arrow annotation callout

1. To create an arrow callout use the following code.
    - **Visual Basic**

```
Private Sub SetUpAnnotations()
    annotationLayer.Annotations.Clear()
    ' Create an arrow callout annotation of polygon type
    Dim contentCenter = New Point(25, -50)
    'Stylise the arrow callout annotation
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
    Dim arrowCallout = New Annotation.Polygon() With {
     .Content = "Low",
     .Style = New ChartStyle() With {
          .Fill = New SolidColorBrush(Colors.Green) With {
              .Opacity = 200.0 / 255
         },
          .Stroke = New SolidColorBrush(Colors.Green)
    },
     .Attachment = AnnotationAttachment.DataIndex,
     .SeriesIndex = 1,
     .PointIndex = 1,
     .ContentCenter = contentCenter,
     .Points = GetPointsForArrowCallout(contentCenter.X, contentCenter.Y, "Low")
}
    ...
```

- **C#**

```
private void SetUpAnnotations()
{
    annotationLayer.Annotations.Clear();
    // Create an arrow callout annotation of polygon type
    var contentCenter = new Point(25, -50);
    var arrowCallout = new Annotation.Polygon()
    {
        Content = "Low",
        //Stylise the arrow callout annotation
        Style = new ChartStyle()
        {
            Fill = new SolidColorBrush(Colors.Green) { Opacity = 200.0 / 255 },
            Stroke = new SolidColorBrush(Colors.Green),
        },
        Attachment = AnnotationAttachment.DataIndex,
        SeriesIndex = 1,
        PointIndex = 1,
        ContentCenter = contentCenter,
        Points = GetPointsForArrowCallout(contentCenter.X, contentCenter.Y, "Low")
    };
    ...
```

2. Define the GetPointsForArrowCallout() method to specify the points for arrow callout.
    1. To measure the size of content string in arrow callout, and reuse it to calculate and set the dimensions of arrow annotation, use the following code.
        - **Visual Basic**

```
Private Function GetPointsForArrowCallout(centerX As Double, centerY As Double, content As String) As PointCollection
    Dim size As _Size = _engine.MeasureString(content)
    Return GetPointsForArrowCallout(centerX, centerY, CSng(size.Width) + 10, CSng(size.Height) + 10)
End Function
```

        - **C#**

```
PointCollection GetPointsForArrowCallout(double centerX, double centerY, string content)
{
    _Size size = _engine.MeasureString(content);
    return GetPointsForArrowCallout(centerX, centerY, (float)size.Width + 10, (float)size.Height + 10);
}
```

    2. To calculate the dimensions and points for arrow annotations, define the method overload GetPointsForArrowCallout() as shown below.
        - **Visual Basic**

```
Private Function GetPointsForArrowCallout(centerX As Double, centerY As Double, rectWidth As Double, rectHeight As Double) As PointCollection
    Dim points = New PointCollection()

    Dim rectLeft As Double = centerX - rectWidth / 2
    Dim rectRight As Double = centerX + rectWidth / 2
    Dim rectTop As Double = centerY - rectHeight / 2
    Dim rectBottom As Double = centerY + rectHeight / 2

    Dim angle As Double = Math.Atan2(-centerY, centerX)
    Dim angleOffset1 As Double = 0.4
    Dim angleOffset2 As Double = 0.04
    Dim arrowHeight As Double = 0.4 * rectHeight
    Dim hypotenuse As Double = arrowHeight / Math.Cos(angleOffset1)
    Dim subHypotenuse As Double = arrowHeight / Math.Cos(angleOffset2)

    Dim isNearBottom As Boolean = Math.Abs(rectTop) > Math.Abs(rectBottom)
    Dim nearHorizontalEdge As Double = If(isNearBottom, rectBottom, rectTop)
    Dim isNearRight As Boolean = Math.Abs(rectLeft) > Math.Abs(rectRight)
    Dim nearVerticalEdge As Double = If(isNearRight, rectRight, rectLeft)
    Dim isHorizontalCrossed As Boolean = Math.Abs(nearHorizontalEdge) > Math.Abs(nearVerticalEdge)
    Dim nearEdge As Double = If(isHorizontalCrossed, nearHorizontalEdge, nearVerticalEdge)

    Dim factor As Integer = If(nearEdge > 0, -1, 1)
    Dim crossedPointOffsetToCenter As Double = If(isHorizontalCrossed, rectHeight / (2 * Math.Tan(angle)) * factor, rectWidth * Math.Tan(angle) * factor / 2)

    ' Arrow points
    points.Add(New Point(0, 0))
    points.Add(New Point(Math.Cos(angle + angleOffset1) * hypotenuse, -Math.Sin(angle + angleOffset1) * hypotenuse))
    points.Add(New Point(Math.Cos(angle + angleOffset2) * subHypotenuse, -Math.Sin(angle + angleOffset2) * subHypotenuse))

    ' Rectangle points
    If isHorizontalCrossed Then
        points.Add(New Point(-nearEdge / Math.Tan(angle + angleOffset2), nearEdge))
        If isNearBottom Then
            points.Add(New Point(rectLeft, rectBottom))
            points.Add(New Point(rectLeft, rectTop))
            points.Add(New Point(rectRight, rectTop))
            points.Add(New Point(rectRight, rectBottom))
        Else
            points.Add(New Point(rectRight, rectTop))
            points.Add(New Point(rectRight, rectBottom))
            points.Add(New Point(rectLeft, rectBottom))
            points.Add(New Point(rectLeft, rectTop))
        End If
        points.Add(New Point(-nearEdge / Math.Tan(angle - angleOffset2), nearEdge))
    Else
        points.Add(New Point(nearEdge, -nearEdge * Math.Tan(angle + angleOffset2)))
        If isNearRight Then
            points.Add(New Point(rectRight, rectBottom))
            points.Add(New Point(rectLeft, rectBottom))
            points.Add(New Point(rectLeft, rectTop))
            points.Add(New Point(rectRight, rectTop))
        Else
            points.Add(New Point(rectLeft, rectTop))
            points.Add(New Point(rectRight, rectTop))
            points.Add(New Point(rectRight, rectBottom))
            points.Add(New Point(rectLeft, rectBottom))
        End If
        points.Add(New Point(nearEdge, -nearEdge * Math.Tan(angle - angleOffset2)))
    End If

    ' Arrow points
    points.Add(New Point(Math.Cos(angle - angleOffset2) * subHypotenuse, -Math.Sin(angle - angleOffset2) * subHypotenuse))
    points.Add(New Point(Math.Cos(angle - angleOffset1) * hypotenuse, -Math.Sin(angle - angleOffset1) * hypotenuse))
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
        Return points
    End Function
```

- **C#**

```
    PointCollection GetPointsForArrowCallout(double centerX, double centerY, double rectWidth, double rectHeight)
    {
        var points = new PointCollection();

        double rectLeft = centerX - rectWidth / 2;
        double rectRight = centerX + rectWidth / 2;
        double rectTop = centerY - rectHeight / 2;
        double rectBottom = centerY + rectHeight / 2;

        double angle = Math.Atan2(-centerY, centerX);
        double angleOffset1 = 0.4;
        double angleOffset2 = 0.04;
        double arrowHeight = 0.4 * rectHeight;
        double hypotenuse = arrowHeight / Math.Cos(angleOffset1);
        double subHypotenuse = arrowHeight / Math.Cos(angleOffset2);

        bool isNearBottom = Math.Abs(rectTop) > Math.Abs(rectBottom);
        double nearHorizontalEdge = isNearBottom ? rectBottom : rectTop;
        bool isNearRight = Math.Abs(rectLeft) > Math.Abs(rectRight);
        double nearVerticalEdge = isNearRight ? rectRight : rectLeft;
        bool isHorizontalCrossed = Math.Abs(nearHorizontalEdge) > Math.Abs(nearVerticalEdge);
        double nearEdge = isHorizontalCrossed ? nearHorizontalEdge : nearVerticalEdge;

        int factor = nearEdge > 0 ? -1 : 1;
        double crossedPointOffsetToCenter = isHorizontalCrossed ?
            rectHeight / (2 * Math.Tan(angle)) * factor : rectWidth * Math.Tan(angle) * factor / 2;

        // Arrow points
        points.Add(new Point(0, 0));
        points.Add(new Point(Math.Cos(angle + angleOffset1) * hypotenuse, -Math.Sin(angle + angleOffset1) * hypotenuse));
        points.Add(new Point(Math.Cos(angle + angleOffset2) * subHypotenuse, -Math.Sin(angle + angleOffset2) * subHypotenuse));

        // Rectangle points
        if (isHorizontalCrossed)
        {
            points.Add(new Point(-nearEdge / Math.Tan(angle + angleOffset2), nearEdge));
            if (isNearBottom)
            {
                points.Add(new Point(rectLeft, rectBottom));
                points.Add(new Point(rectLeft, rectTop));
                points.Add(new Point(rectRight, rectTop));
                points.Add(new Point(rectRight, rectBottom));
            }
            else
            {
                points.Add(new Point(rectRight, rectTop));
                points.Add(new Point(rectRight, rectBottom));
                points.Add(new Point(rectLeft, rectBottom));
                points.Add(new Point(rectLeft, rectTop));
            }
            points.Add(new Point(-nearEdge / Math.Tan(angle - angleOffset2), nearEdge));
        }
        else
        {
            points.Add(new Point(nearEdge, -nearEdge * Math.Tan(angle + angleOffset2)));
            if (isNearRight)
            {
                points.Add(new Point(rectRight, rectBottom));
                points.Add(new Point(rectLeft, rectBottom));
                points.Add(new Point(rectLeft, rectTop));
                points.Add(new Point(rectRight, rectTop));
            }
            else
            {
                points.Add(new Point(rectLeft, rectTop));
                points.Add(new Point(rectRight, rectTop));
                points.Add(new Point(rectRight, rectBottom));
                points.Add(new Point(rectLeft, rectBottom));
            }
            points.Add(new Point(nearEdge, -nearEdge * Math.Tan(angle - angleOffset2)));
        }

        // Arrow points
        points.Add(new Point(Math.Cos(angle - angleOffset2) * subHypotenuse, -Math.Sin(angle - angleOffset2) * subHypotenuse));
        points.Add(new Point(Math.Cos(angle - angleOffset1) * hypotenuse, -Math.Sin(angle - angleOffset1) * hypotenuse));
        return points;
    }
```

Back to Top

Step 3: Render the annotations in chart

To Render the annotations in chart, follow these steps:

1. Define global field of render engine.
   - **Visual Basic**

```
Dim _engine As IRenderEngine
```

   - **C#**

```
IRenderEngine _engine;
```

2. To create an instance of AnnotationLayer use the following code.

XAML copyCode

```
<c1:C1FlexChart.Layers>
    <c1:AnnotationLayer x:Name="annotationLayer" />
</c1:C1FlexChart.Layers>
```

3. To add the annotation callouts in annotationLayer, use the following code.
   - **Visual Basic**

```
    annotationLayer.Annotations.Add(arrowCallout)
    annotationLayer.Annotations.Add(lineCallout)
End Sub
```

   - **C#**

```
    annotationLayer.Annotations.Add(arrowCallout);
    annotationLayer.Annotations.Add(lineCallout);
}
```

4. To render the callouts use the following code in the Rendered event of chart.
   - **Visual Basic**

```
Private Sub flexChart_Rendered(sender As Object, e As C1.WPF.Chart.RenderEventArgs) Handles flexChart.Rendered
    If _engine Is Nothing Then
        _engine = e.Engine
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
            SetUpAnnotations()
        End If
    End Sub
```

- **C#**

```
private void flexChart_Rendered(object sender, C1.WPF.Chart.RenderEventArgs e)
{
    if (_engine == null)
    {
        _engine = e.Engine;
        SetUpAnnotations();
    }
}
```

Back to Top

FlexChart Legend

Legends are important aspects of charts. They are key to better understand and interpret information conveyed by charts. FlexChart provides API members to customize the appearance and behavior of the chart legend.

The following topics demonstrate how to customize the FlexChart legend.

- Legend Position
- Legend Style
- Legend Toggle
- Legend Text Wrap
- Legend Grouping

Legend Position

You can use the LegendPosition property to position the Legend relative to the Plot Area, as per your requirements.

The LegendPosition property can be set to any of the following values in the Position enumeration:

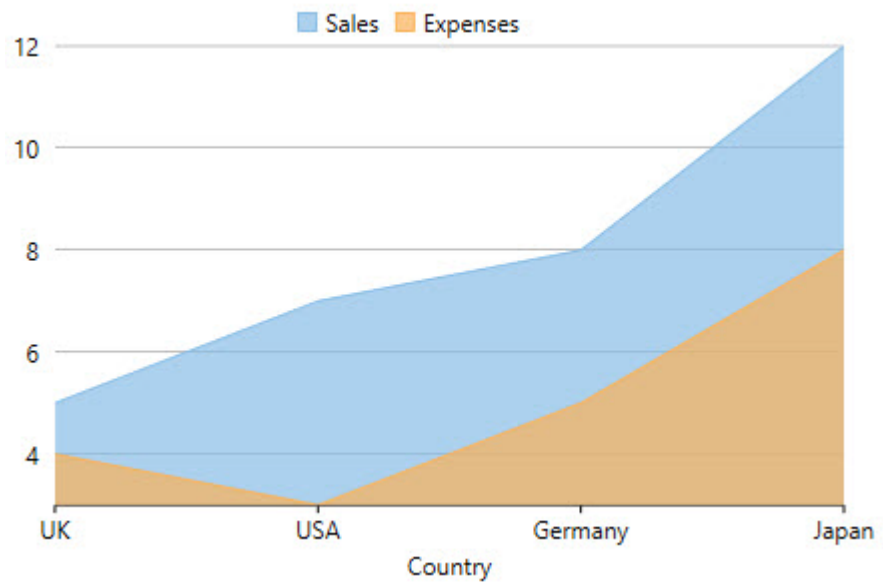
Property	Description
Position.Auto	Positions the legend automatically.
Position.Bottom	Positions the legend below the plot.
Position.Left	Positions the legend to the left of the plot.
Position.None	Hides the legend.
Position.Right (default value)	Positions the legend to the right of the plot.
Position.Top	Positions the legend above the plot.

Here is the code snippet for setting the property:

- **C#**

```
flexChart.LegendPosition = C1.Chart.Position.Top;
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.



Legend Style

FlexChart lets you can customize the Legend using the LegendStyle property.

The table below lists the properties available for customizing the Legend:

Property	Description
Fill	Specifies the fill color.
FontSize	Sets the font of the Legend.
FontStyle	Sets the font of the Legend.
FontWeight	Sets the font of the Legend.
FontFamily	Sets the font of the Legend.
Stroke	Sets the stroke color.
StrokeThickness	Sets the stroke width.

Legend Toggle

FlexChart allows you to toggle the visibility of a series in the plot, when you click the series item in the legend by means of the LegendToggle property.

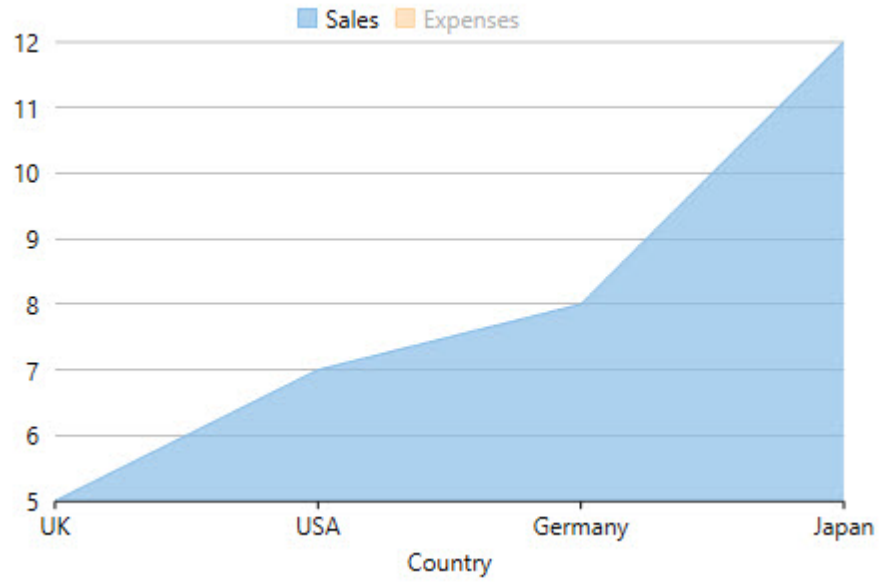
The default value of the **LegendToggle** property is False. To enable series toggling, you need to set the **LegendToggle** property to True.

Here is the code snippet:

- **C#**

```
flexChart.LegendToggle = true;
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.



Legend Text Wrap

Legend text wrap is a feature to shorten the legend entries by either truncating or wrapping them into multiple lines. This feature gives user the flexibility to effectively utilize chart display area by adjusting the space occupied by legends.

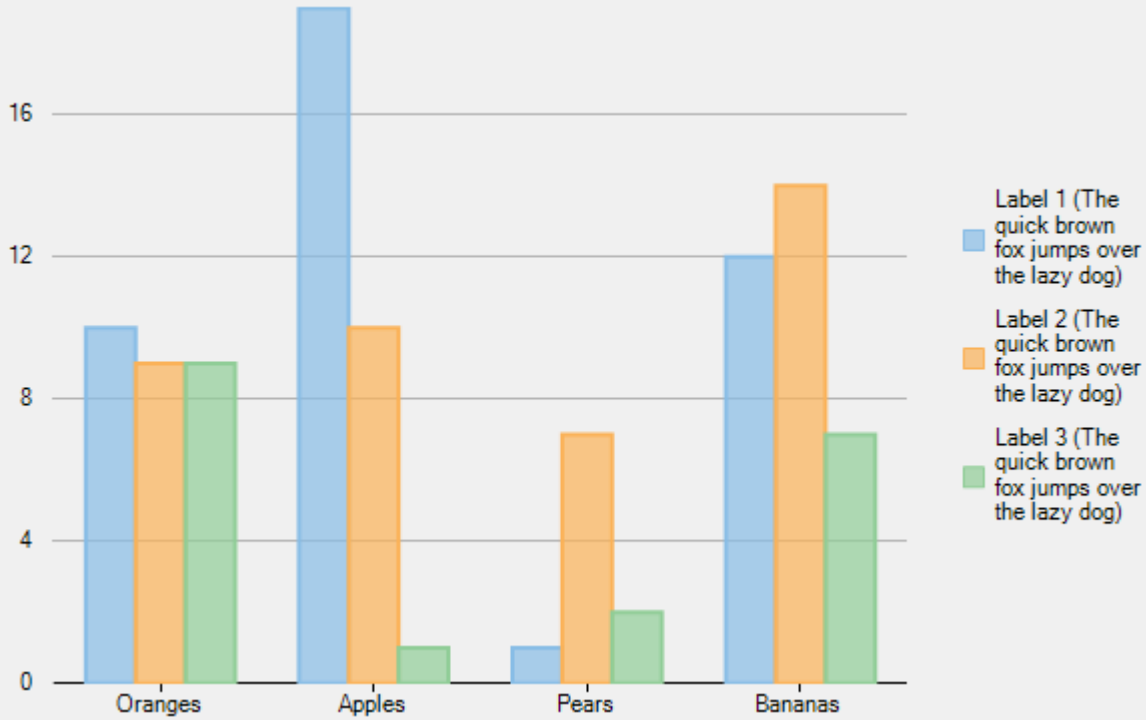
FlexChart provides text wrapping for the legend text that exceeds the value specified in LegendMaxWidth property, which sets the maximum width of each legend entry. The control enables you to manage legend entries in following two ways:

- **Wrap**: This mode allows you to wrap or break the legend entries into multiple lines. Wrapping text is useful when you need the entire legend text to be visible in chart area. To wrap legend texts in FlexChart, set the LegendTextWrapping property to **Wrap**.
- **Truncate**: This mode allows you to shorten legend entries by cutting off the text from the end. If you want to truncate legend texts in FlexChart, set the LegendTextWrapping property to **Truncate**.

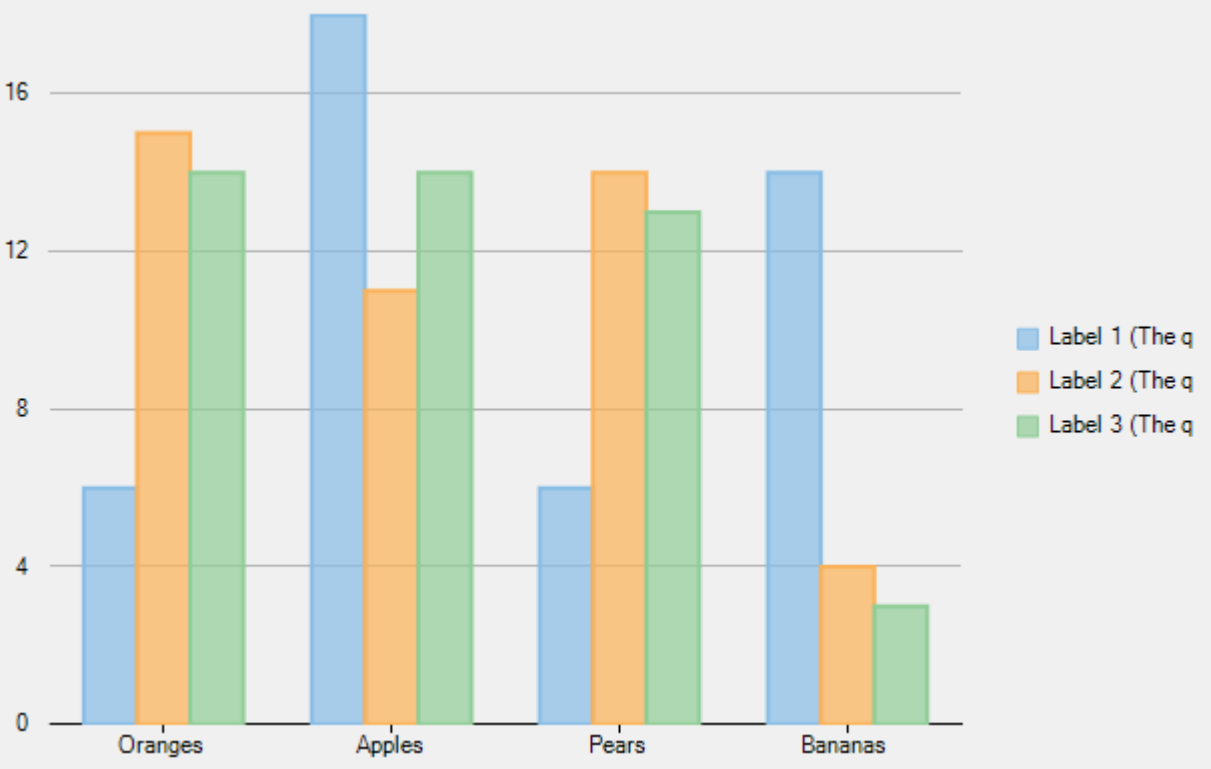
In FlexChart, the maximum width set for the legend entries affects both text wrapping and text truncating. The greater the value set for maximum legend entry width, the less the legend text is wrapped or truncated.

The following image displays legend texts wrapped into multiple lines.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



The following image displays truncated legend texts.



The following code compares fruit data for three consecutive months. The code shows how to implement legend text wrapping in FlexChart.

- **XAML**

```
<Chart:C1FlexChart x:Name="flexChart"
                   ItemsSource="{Binding DataContext.Data}"
                   BindingX="Fruit"
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
                    LegendTextWrapping="Wrap"
                    LegendMaxWidth="80"
                    LegendPosition="Right">
    <Chart:Series SeriesName="系列1 （凡例テキストを折り返す）"
                  Binding="March"/>
    <Chart:Series SeriesName="系列2 （凡例テキストを折り返す）"
                  Binding="April"/>
    <Chart:Series SeriesName="系列3 （凡例テキストを折り返す）"
                  Binding="May"/>
</Chart:C1FlexChart>
```

- **Visual Basic**

```
flexChart.LegendTextWrapping = C1.Chart.TextWrapping.Wrap
flexChart.LegendPosition = C1.Chart.Position.Right
flexChart.LegendMaxWidth = 80
```

- **C#**

```
flexChart.LegendTextWrapping = C1.Chart.TextWrapping.Wrap;
flexChart.LegendPosition = C1.Chart.Position.Right;
flexChart.LegendMaxWidth = 80;
```

Legend Grouping

Legend group, as the name suggests, categorizes the legend entries of chart series based on the data represented by them. So, the multiple chart series with similar data can be better presented as groups in legend. This organizes the legends, which helps in better visualization and analysis of charts depicting multiple series.

FlexChart supports grouping respective legend items of different series in chart through LegendGroup property exposed by the **Series** class. By setting the **LegendGroup** property to a string value, you can specify the group name to which a particular series or legend item belongs. The series for which value of LegendGroup property is same are grouped together in the legend. However, if the LegendGroup property is not defined for a series then the series becomes a part of 0th group.

The value of **LegendGroup** property gets displayed as group title above the corresponding legend items. However, legend items that belong to the 0th group get displayed without any group title.

Positioning Legend Groups

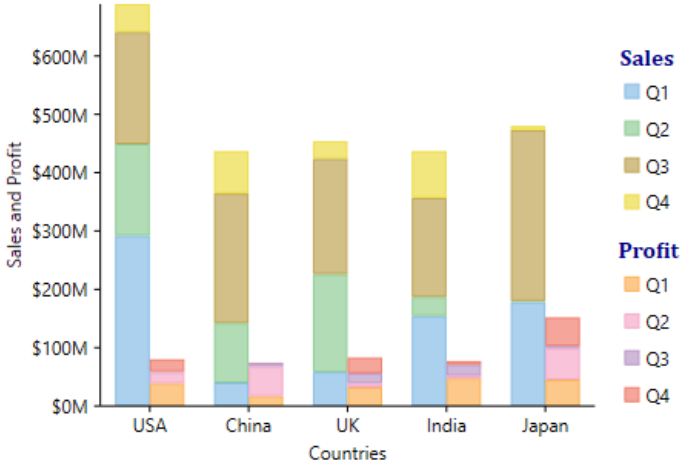
The legend groups get positioned automatically with respect to each other depending on legend's position. For example, if the legends are positioned on top or bottom of the chart, then the legend groups are stacked horizontally one alongside the other. Whereas, if the legends are positioned to left or right of the chart, then the legend groups are stacked vertically one above the other.

Styling Legend Groups

FlexChart also supports styling and formatting of the legend group titles. The appearance of legend group titles can be customized by specifying the LegendGroupHeaderStyle property.

Following image shows a stacked chart that plots country-wise sales and profit of a company for different quarters of a year. Here, the legend items have been grouped together as per the stacked series for quick and easy analysis. The image also shows how legend and the legend groups have got positioned vertically and how appearance of the group titles can be customized.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



Following code snippet demonstrates how to group the legends of respective series together by setting the Series.LegendGroup property of those series to the desired group name. The code snippet also shows how GroupHeaderStyle property can be used to style headers of the legend groups.

- **Xaml**

```
<Chart:C1FlexChart x:Name="flexChart" ItemsSource="{Binding DataContext.Data}"
                   BindingX="Country" Stacking="Stacked">
    <!--Grouping the legend items-->
    <Chart:Series SeriesName="Q1" Binding="SalesQ1" LegendGroup="Sales" StackingGroup="0" />
    <Chart:Series SeriesName="Q1" Binding="ProfitQ1" LegendGroup="Profit" StackingGroup="1" />
    <Chart:Series SeriesName="Q2" Binding="SalesQ2" LegendGroup="Sales" StackingGroup="0" />
    <Chart:Series SeriesName="Q2" Binding="ProfitQ2" LegendGroup="Profit" StackingGroup="1" />
    <Chart:Series SeriesName="Q3" Binding="SalesQ3" LegendGroup="Sales" StackingGroup="0" />
    <Chart:Series SeriesName="Q3" Binding="ProfitQ3" LegendGroup="Profit" StackingGroup="1" />
    <Chart:Series SeriesName="Q4" Binding="SalesQ4" LegendGroup="Sales" StackingGroup="0" />
    <Chart:Series SeriesName="Q4" Binding="ProfitQ4" LegendGroup="Profit" StackingGroup="1" />

    <Chart:C1FlexChart.AxisY>
        <Chart:Axis Format="$0M"
                    Labels="True"
                    Title="Sales and Profit"/>
    </Chart:C1FlexChart.AxisY>

    <Chart:C1FlexChart.AxisX>
        <Chart:Axis Title="Countries"/>
    </Chart:C1FlexChart.AxisX>

    <!--Styling the legend group headers-->
<Chart:C1FlexChart.LegendGroupHeaderStyle>
    <Chart:ChartStyle Stroke="DarkBlue" FontFamily="Cambria" FontStyle="Normal" FontSize="15" FontWeight="Bold"/>
</Chart:C1FlexChart.LegendGroupHeaderStyle>
</Chart:C1FlexChart>
```

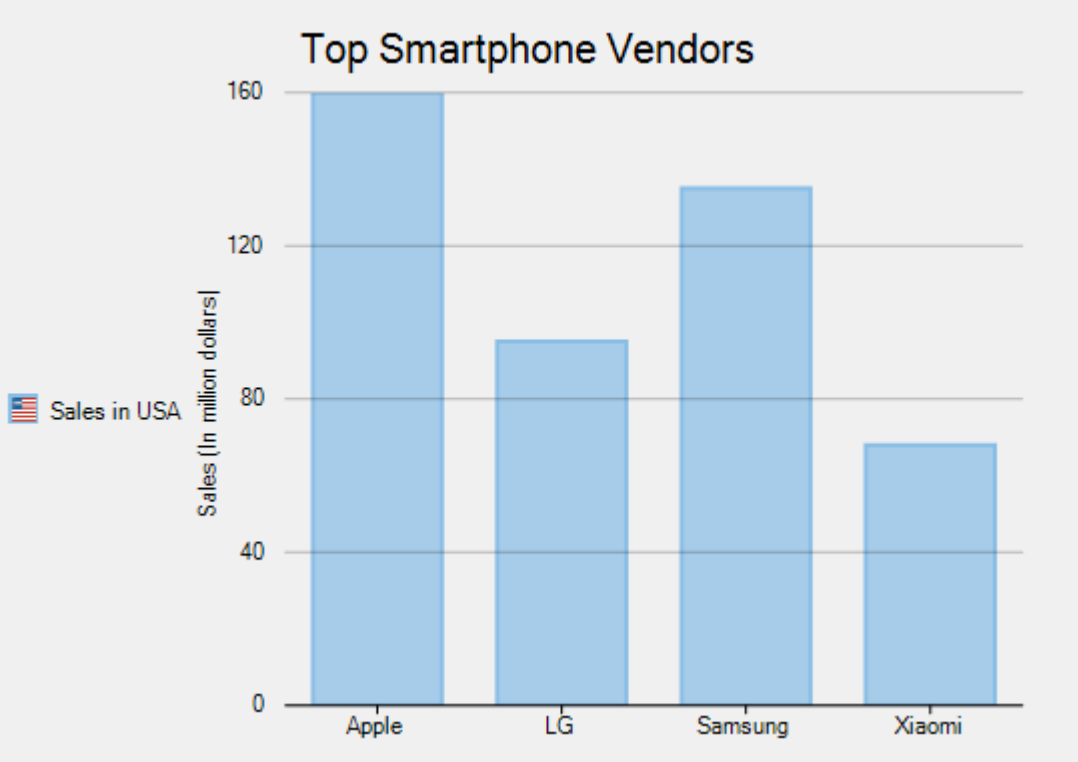
Custom Legend Icon

FlexChart allows you to apply custom image or icon for customizing the legend items. To enable FlexChart to display custom legend icon, implement the GetLegendItemImageSource method provided by ISeries interface. This method primarily accepts two parameters; index and _size. The index parameter refers to the legend item position and _size parameter refers to the default legend icon size.

In the example code below, we have implemented the GetLegendItemImageSource method to customize the image size and draw a border around it. This method then returns the image object. To apply the custom legend icon add object of the class SeriesWithPointLegendItems to the chart Series collection.

The image shows how FlexChart appears after using custom legend icon.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



Use the following code snippet to implement custom legend icon.

XAML

```
<c1:C1FlexChart x:Name="flexChart" ItemsSource="{Binding SmartPhoneVendors}"
Binding="Sales" BindingX="Name" Header="Top Smartphone Vendors" Grid.Row="1">
            <c1:C1FlexChart.HeaderStyle>
                <c1:ChartStyle FontSize="15" FontFamily="GenericSansSerif"/>
            </c1:C1FlexChart.HeaderStyle>
</c1:C1FlexChart>
```

Code

HTML

```
public partial class LegendItems
    {
        static List<SmartPhoneVendor> vendors = new List<SmartPhoneVendor>();
        static Image LegendIconImage = Properties.Resources.usa;
        Series customSeries;
        public LegendItems()
        {
            InitializeComponent();
            vendors = SmartPhoneVendors();

            //Add custom series
            customSeries = new SeriesWithPointLegendItems();
            customSeries.Name = "Sales in USA";
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
            flexChart1.Series.Add(customSeries);
            flexChart1.Legend.Position = Position.Left;
            flexChart1.ToolTip.Content = "{seriesName}\r\n{value}";

        }
        public class SeriesWithPointLegendItems : Series, ISeries
        {
            object ISeries.GetLegendItemImageSource(int index, ref C1.Chart._Size
imageSize)
            {
                {
                    // Original images/logos are all 50x50 pixels.
                    // Here they are replaced with new images where a 5 pixel border
is added
                    // around the logos.
                    imageSize.Height = 60;
                    imageSize.Width = 60;

                    SmartPhoneVendor vendor = vendors.ElementAt(index);
                    if (LegendIconImage != null && LegendIconImage.Width != 60)
                    {
                        Bitmap bmp = new Bitmap(60, 60);
                        using (SolidBrush sb = new SolidBrush(vendor.Color))
                        {
                            using (Graphics g = Graphics.FromImage(bmp))
                            {
                                Rectangle r = new Rectangle(0, 0,
(int)imageSize.Width, (int)imageSize.Height);
                                using (Pen p = new Pen(sb))
                                {
                                    g.DrawRectangle(p, r);
                                }
                                g.FillRectangle(sb, r);

                                Point ci = new Point((int)(0.5 * (imageSize.Width -
LegendIconImage.Width)),
                                    (int)(0.5 * (imageSize.Height -
LegendIconImage.Height)));
                                g.DrawImage(LegendIconImage, ci);
                            }
                        }
                        LegendIconImage = bmp;
                    }
                    // Keep the original size of the logo bitmaps, but reduce their
size if the chart window
                    // is too small to display the bitmaps properly.
                    Size bounds = this.Chart.ClientSize;
                    double divadj = (bounds.Height > 800) ? 12 : 25;
                    double fracHeight = bounds.Height / divadj;
                    if (fracHeight < imageSize.Height)
                        imageSize.Width = imageSize.Height = fracHeight;
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
                return LegendIconImage;
            }
        }
    }
    private static List<SmartPhoneVendor> SmartPhoneVendors()
    {

        vendors.Add(new SmartPhoneVendor()
        {
            Name = "Apple",
            Color = Color.FromArgb(136, 189, 230),
            Sales = 350,
        });
        vendors.Add(new SmartPhoneVendor()
        {
            Name = "LG",
            Color = Color.FromArgb(251, 178, 88),
            Sales = 120,
        });

        vendors.Add(new SmartPhoneVendor()
        {
            Name = "Samsung",
            Color = Color.FromArgb(188, 153, 199),
            Sales = 280,
        });

        vendors.Add(new SmartPhoneVendor()
        {
            Name = "Xiaomi",
            Color = Color.FromArgb(240, 126, 110),
            Sales = 68,
        });

        return vendors;
    }
    public class SmartPhoneVendor
    {
        public string Name { get; set; }
        public double Sales { get; set; }
        public Color Color { get; set; }
    }
}
```

FlexChart Series

A series is a set of data or more specifically related data points that are plotted on a chart.

In FlexChart, a series is represented by the Series object, which provides the entire data plotted on the chart. And the Flexchart.Series collection comprises all data series (Series objects) in the control.

You can assign any of the following properties to a series in FlexChart:

Copyright © 2019 GrapeCity, Inc. All rights reserved.

- An X-axis (Series.AxisX)
- A Y-axis (Series.AxisY)
- A property containing Y values for the series (Series.Binding)
- A property containing X values for the series (Series.BindingX)
- A chart type (Series.ChartType)
- A collection of objects containing the series data (Series.ItemsSource)
- A name (Series.SeriesName)

A series consists of a collection of data points that you can customize using the following properties:

- To set the shape of the marker to be used for each data point in the series (Series.SymbolMarker)
- To set the size of the symbol used to render the series (Series.SymbolSize)
- To set the symbol style used in the data points in the series (**Series.SymbolStyle**)

Once you have set these properties in a series, the same settings are inherited by all data points.

Here are the links to key information regarding the Series object in FlexChart:

- Creating and Adding a Series
- Adding Data to Series
- Emphasizing Different Types of Data
- Customizing Series

Creating and Adding Series

By default, **FlexChart for WPF** displays three series containing dummy data at design-time as well as run-time. However, you can provide your own data and display the series with that data. For information on how to provide data to FlexChart, refer to Providing Data.

FlexChart enables you to create and add a series at design-time as well as run-time. So if you want to add more series, you can do so by performing any of the methods mentioned below.

At Design-Time

Perform the following steps to create and add a series in FlexChart at design-time:

1. In the **Properties** window, click the **Miscellaneous** drop down.
2. Navigate to the **Series** field.
3. Click the **Ellipsis** button next to the **Series** field.
   Series Collection Editor: Series appears.

   > By default, FlexChart contains three series added in the Series collection. Thus, **Series Collection Editor** appears with the pre-added series.
4. Click the **Add** button to add an additional series in the Series collection.
5. Repeat step 3 to add the required number of series.
6. Click the **OK** button.

At Run-Time

At run-time, you first need to create a series by using the **Series** object. And then, you need to add the series to the FlexChart Series collection using the **Add** method in the **FlexChart.Series** collection property.

The following code shows how to create and add a series in FlexChart at run-time.

XAML

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
<c1:Series AxisX="{x:Null}"
           AxisY="{x:Null}"
           Binding="Y"
           BindingX="X"
           Chart="{x:Null}"
           SeriesName="Series 4">
    <c1:Series.ItemsSource>
        <PointCollection>1,16 2,19 3,15 4,22 5,18</PointCollection>
    </c1:Series.ItemsSource>
</c1:Series>
```

Code

C# copyCode

```
C1.WPF.Chart.Series series4 = new C1.WPF.Chart.Series();
flexChart.Series.Add(series4);
```

Adding Data to Series

When it comes to adding data to series, FlexChart provides a powerful way through binding. You can bind series in FlexChart with multiple data sources, which enables you to combine data from multiple data sources. To plot data from multiple data sources, you need to use the Series.ItemsSource property.

See the following code for reference. Notice that the following code user the **DataCreator.cs** class to generate data.

XAML

```
<c1:C1FlexChart x:Name="flexChart"
                ItemsSource="{Binding DataContext.Data}"
                ChartType="Scatter">
    <c1:C1FlexChart.Series>
        <c1:Series  x:Name="Function1"
                    SeriesName="Function1"
                    BindingX ="XVals"
                    Binding="YVals"/>
        <c1:Series  x:Name="Function2"
                    SeriesName="Function2"
                    BindingX ="XVals"
                    Binding="YVals"/>
    </c1:C1FlexChart.Series>
</c1:C1FlexChart>
```

Code

DataCreator.cs copyCode

```
class DataCreator
{
    public delegate double MathActionDouble(double num);
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
    public delegate double MathActionInt(int num);

    public static List<DataPoint> Create(MathActionDouble function, double from,
double to, double step)
    {
        var result = new List<DataPoint>();
        var count = (to - from) / step;

        for (double r = from; r < to; r += step)
        {
            result.Add(new DataPoint()
            {
                XVals = r,
                YVals = function(r)
            });
        }
        return result;
    }

    public static List<DataPoint> Create(MathActionInt function, int from, int to,
int step)
    {
        var result = new List<DataPoint>();
        var count = (to - from) / step;

        for (int r = from; r < to; r += step)
        {
            result.Add(new DataPoint()
            {
                XVals = r,
                YVals = function(r)
            });
        }
        return result;
    }

    public static List<DataPoint> Create(MathActionDouble functionX, MathActionDouble
functionY, int ptsCount)
    {
        var result = new List<DataPoint>();

        for (double i = 0; i < ptsCount; i++)
        {
            result.Add(new DataPoint()
            {
                XVals = functionX(i),
                YVals = functionY(i)
            });
        }
        return result;
    }
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

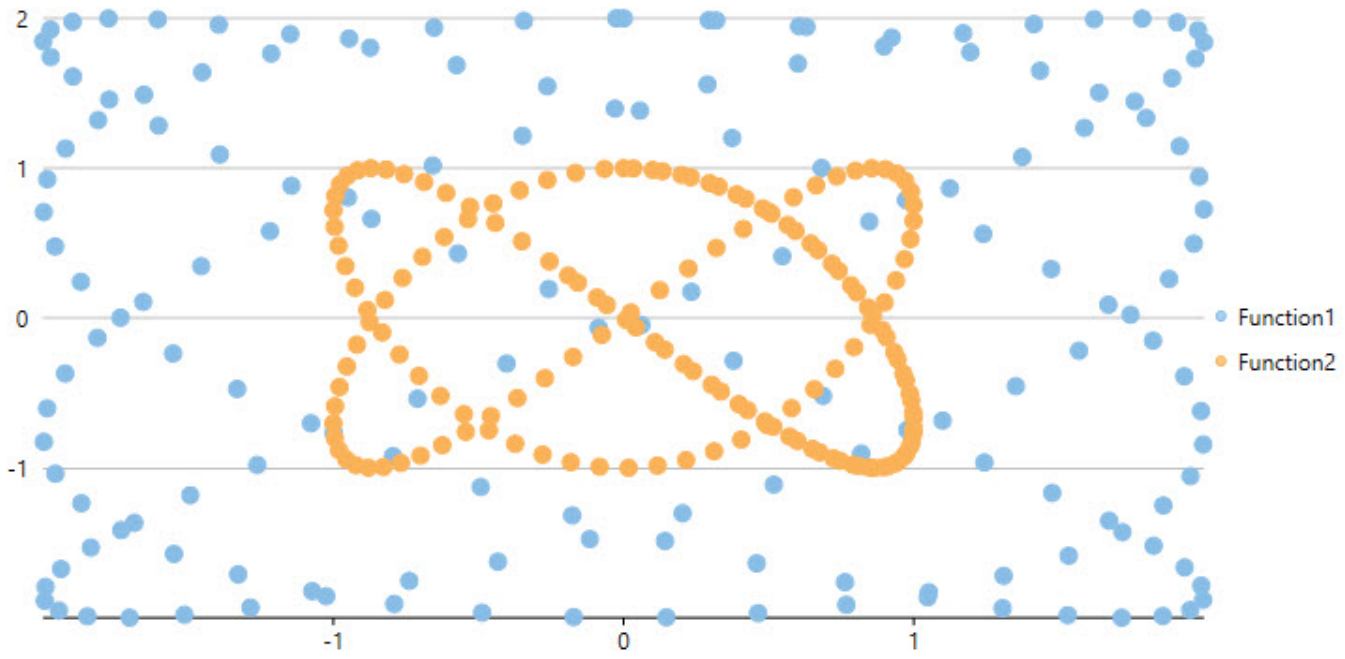
```
}

public class DataPoint
{
    public double XVals { get; set; }
    public double YVals { get; set; }
}
```

MainWindow.xaml.cs copyCode

```
public partial class MainWindow : Window
{


    List<DataPoint> _function1Source;
    List<DataPoint> _function2Source;

    public MainWindow()
    {
        this.InitializeComponent();
        this.Loaded += Form_Loaded;
    }

    private void Form_Loaded(object sender, EventArgs e)
    {
        SetupChart();
    }

    void SetupChart()
    {
        flexChart.BeginUpdate();
        this.Function1.ItemsSource = Function1Source;
        this.Function2.ItemsSource = Function2Source;
        flexChart.EndUpdate();
    }

    public List<DataPoint> Function1Source
    {
        get
        {
            if (_function1Source == null)
            {
                _function1Source = DataCreator.Create(x => 2 * Math.Sin(0.16 * x), y
=> 2 * Math.Cos(0.12 * y), 160);
            }

            return _function1Source;
        }
    }

    public List<DataPoint> Function2Source
    {
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
        get
        {
            if (_function2Source == null)
            {
                _function2Source = DataCreator.Create(x => Math.Sin(0.1 * x), y =>
Math.Cos(0.15 * y), 160);
            }

            return _function2Source;
        }
    }
}
```



Emphasizing Different Types of Data

In a chart, there is often a common but crucial requirement to emphasize different types of data. Emphasizing or distinguishing different types of data is vital because chart data that is distinguishable is easier to interpret and understand.

FlexChart caters to this requirement by allowing you to combine two or more chart types in a single chart. For instance, you can combine the LineSymbols Chart with the Column Chart to make the chart data easier to interpret. You can use the ChartType property to specify the chart type at the series level for each series, thereby creating charts with multiple chart types.

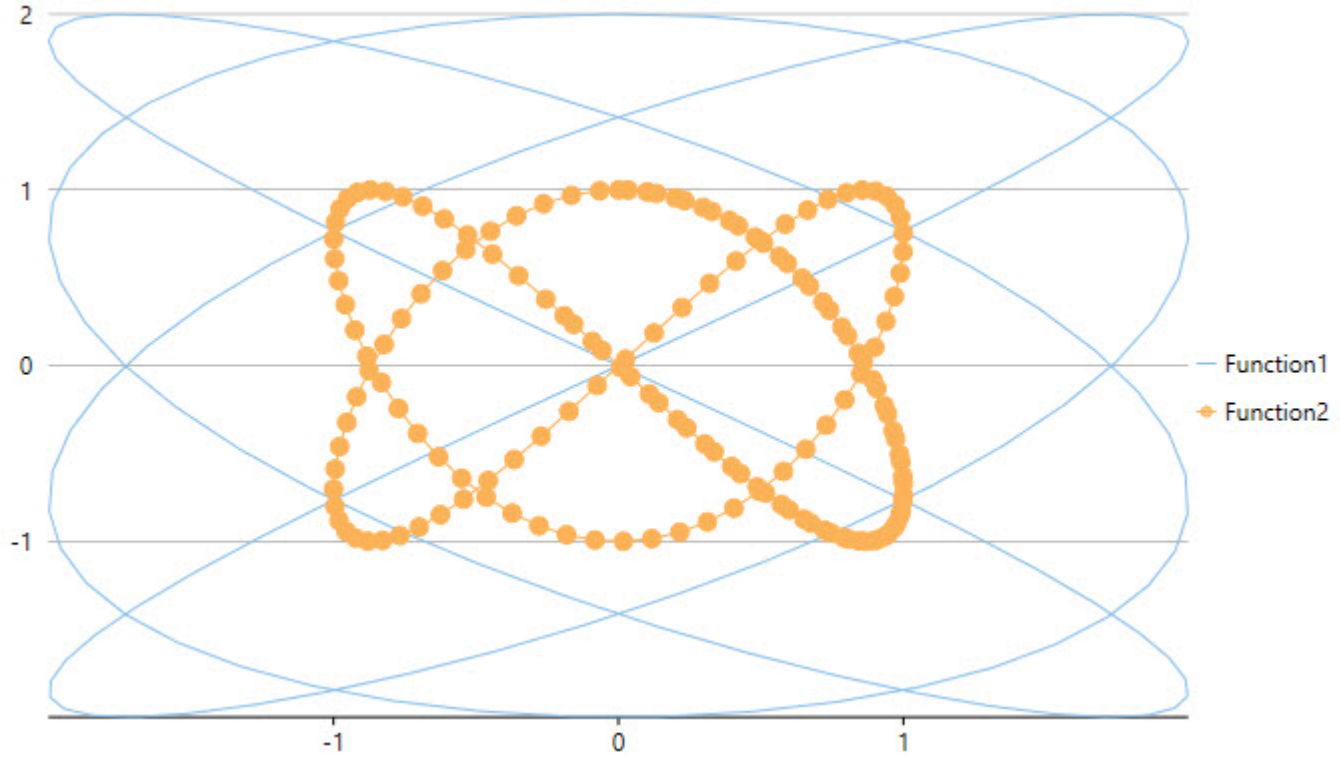
The following code combines two chart types in a single chart.

XAML

```
<c1:C1FlexChart.Series>
    <c1:Series ChartType="Line"
               x:Name="Function1"
               SeriesName="Function1"
               BindingX ="XVals"
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
                Binding="YVals"/>
    <c1:Series ChartType="LineSymbols"
                x:Name="Function2"
                SeriesName="Function2"
                BindingX ="XVals"
                Binding="YVals"/>
</c1:C1FlexChart.Series>
```



Customizing Series

Once the series have been displayed in the chart, you can customize the displayed series to manage the same more efficiently.

FlexChart allows you to customize series by showing or hiding a series either in the Plot Area or the Legend or both. If there are hundreds of series to be displayed in your chart, you would certainly need to manage the same due to the space limitation of the chart.

In FlexChart, you can manage series by using the Visibility property of a series. The Visibility property accepts values of the SeriesVisibility enumerated type.

You can set the property to the following different values to show or hide a series:

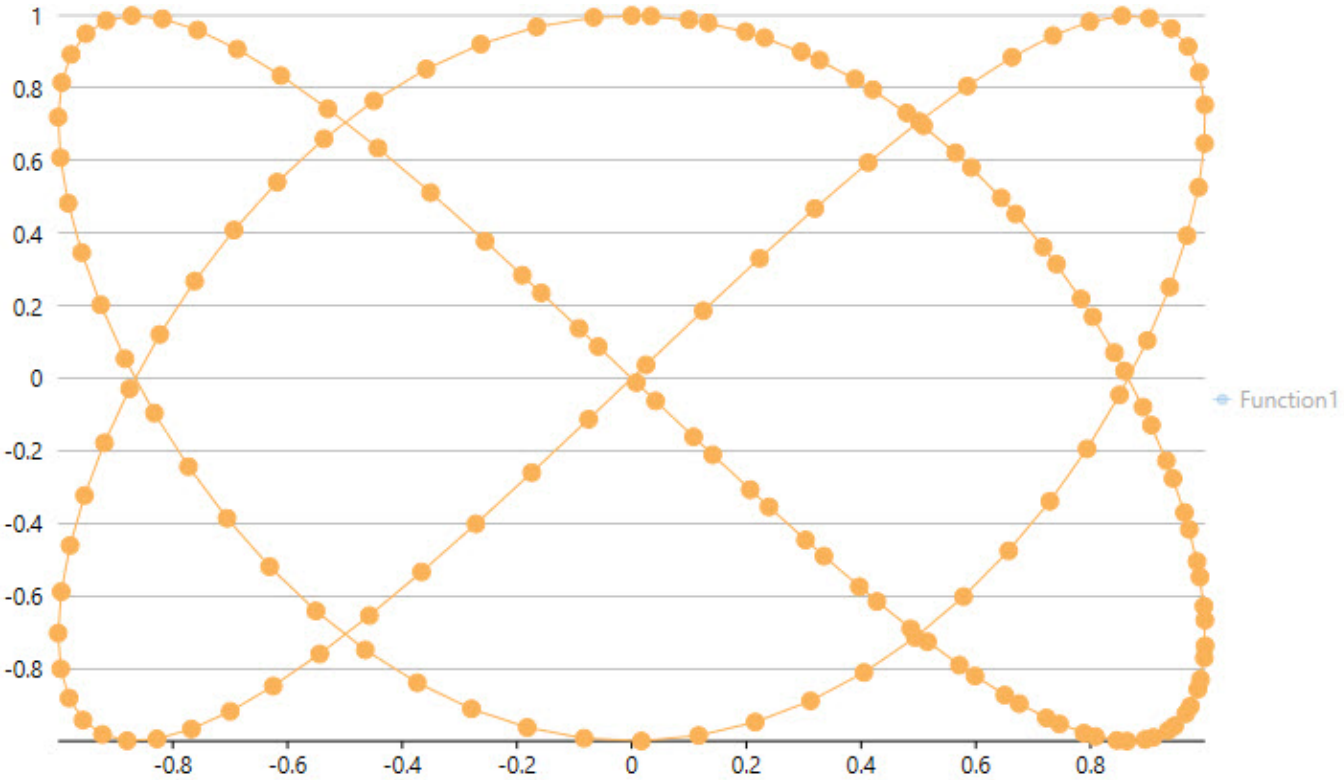
Value	Description
SeriesVisibility.Visible	The series is displayed in the Plot as well as the Legend.
SeriesVisibility.Plot	The series is displayed in the Plot, but hidden in the Legend.
SeriesVisibility.Legend	The series is displayed in the Legend, but hidden in the Plot.
SeriesVisibility.Hidden	The series is hidden in the Plot as well as the Legend.

Here's the code snippet for reference:

XAML

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
<c1:C1FlexChart.Series>
    <c1:Series Visibility="Legend"
               x:Name="Function1"
               SeriesName="Function1"
               BindingX ="XVals"
               Binding="YVals"/>
    <c1:Series Visibility="Plot"
               x:Name="Function2"
               SeriesName="Function2"
               BindingX ="XVals"
               Binding="YVals"/>
</c1:C1FlexChart.Series>
```



In addition, you can enhance the visual appeal of the series by setting different palettes for FlexChart. For more details, refer to Setting FlexChart Palette.

You can also work with different symbol styles to render visually appealing series in the chart. For more information, refer to Symbol styles for Series.

Box-and-Whisker

Box-and-Whisker series allows you to display groups of data into the range, quartiles, and median. The name itself suggests that the series depicts data through boxes and whiskers.

A box is the range showing the quartiles (lower and upper) and the median. Whiskers, on the other hand, are the lines extending vertically from the boxes. These lines indicate the data variability outside the lower and the upper quartiles. In addition, points that lie outside of these lines are known as outliers.

Box-and-Whisker series is ideal for visualizing statistical distribution or examining multiple sets of data graphically.

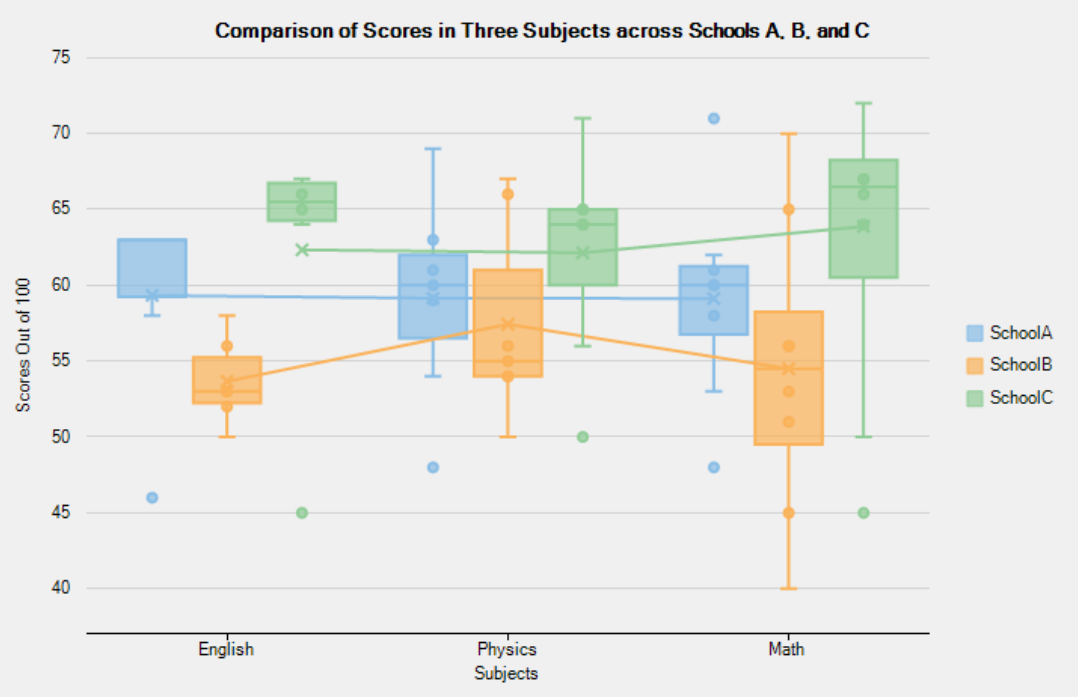
Box-and-Whisker series in FlexChart allows working with different features, as follows:

- **Quartile**: Specify whether you would like to calculate quartiles by including or excluding median. To specify quartile calculation, set the QuartileCalculation property from the QuartileCalculation enumeration.
- **Inner points**: Indicate whether to show or hide inner points by setting the ShowInnerPoints property.

Copyright © 2019 GrapeCity, Inc. All rights reserved.

- **Outliers**: Indicate whether to show outliers by setting the ShowOutliers property.
- **Mean line**: Display the mean line by setting the ShowMeanLine property.
- **Mean marks**: Show mean marks by setting the ShowMeanMarks property.

The following image displays quartiles, median, and whiskers for the data that compares scores of students in three subjects across different schools.



The following code uses data regarding scores obtained by students of schools A, B, and C in three subjects. The code illustrates how to implement Box-and-Whisker series in FlexChart.

- **DataCreator.vb**

```
Class DataCreator
    Public Shared Function CreateSchoolScoreData() As List(Of ClassScore)
        Dim result = New List(Of ClassScore)()
        result.Add(New ClassScore() With {
            .ClassName = "English",
            .SchoolA = 46,
            .SchoolB = 53,
            .SchoolC = 66
        })
        result.Add(New ClassScore() With {
            .ClassName = "Physics",
            .SchoolA = 61,
            .SchoolB = 55,
            .SchoolC = 65
        })
        result.Add(New ClassScore() With {
            .ClassName = "English",
            .SchoolA = 58,
            .SchoolB = 56,
            .SchoolC = 67
        })
        result.Add(New ClassScore() With {
            .ClassName = "Math",
            .SchoolA = 58,
            .SchoolB = 51,
            .SchoolC = 64
        })
        result.Add(New ClassScore() With {
            .ClassName = "English",
            .SchoolA = 63,
            .SchoolB = 53,
            .SchoolC = 45
        })
        result.Add(New ClassScore() With {
            .ClassName = "English",
            .SchoolA = 63,
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
    .SchoolB = 50,
    .SchoolC = 65
})
result.Add(New ClassScore() With {
    .ClassName = "Math",
    .SchoolA = 60,
    .SchoolB = 45,
    .SchoolC = 67
})
result.Add(New ClassScore() With {
    .ClassName = "Math",
    .SchoolA = 62,
    .SchoolB = 53,
    .SchoolC = 66
})
result.Add(New ClassScore() With {
    .ClassName = "Physics",
    .SchoolA = 63,
    .SchoolB = 54,
    .SchoolC = 64
})
result.Add(New ClassScore() With {
    .ClassName = "English",
    .SchoolA = 63,
    .SchoolB = 52,
    .SchoolC = 67
})
result.Add(New ClassScore() With {
    .ClassName = "Physics",
    .SchoolA = 69,
    .SchoolB = 66,
    .SchoolC = 71
})
result.Add(New ClassScore() With {
    .ClassName = "Physics",
    .SchoolA = 48,
    .SchoolB = 67,
    .SchoolC = 50
})
result.Add(New ClassScore() With {
    .ClassName = "Physics",
    .SchoolA = 54,
    .SchoolB = 50,
    .SchoolC = 56
})
result.Add(New ClassScore() With {
    .ClassName = "Physics",
    .SchoolA = 60,
    .SchoolB = 56,
    .SchoolC = 64
})
result.Add(New ClassScore() With {
    .ClassName = "Math",
    .SchoolA = 71,
    .SchoolB = 65,
    .SchoolC = 50
})
result.Add(New ClassScore() With {
    .ClassName = "Math",
    .SchoolA = 48,
    .SchoolB = 70,
    .SchoolC = 72
})
result.Add(New ClassScore() With {
    .ClassName = "Math",
    .SchoolA = 53,
    .SchoolB = 40,
    .SchoolC = 80
})
result.Add(New ClassScore() With {
    .ClassName = "Math",
    .SchoolA = 60,
    .SchoolB = 56,
    .SchoolC = 67
})
result.Add(New ClassScore() With {
    .ClassName = "Math",
    .SchoolA = 61,
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
            .SchoolB = 56,
            .SchoolC = 45
        })
        result.Add(New ClassScore() With {
            .ClassName = "English",
            .SchoolA = 63,
            .SchoolB = 58,
            .SchoolC = 64
        })
        result.Add(New ClassScore() With {
            .ClassName = "Physics",
            .SchoolA = 59,
            .SchoolB = 54,
            .SchoolC = 65
        })

        Return result
    End Function
End Class

Public Class ClassScore
    Public Property ClassName() As String
        Get
            Return m_ClassName
        End Get
        Set
            m_ClassName = Value
        End Set
    End Property
    Private m_ClassName As String
    Public Property SchoolA() As Double
        Get
            Return m_SchoolA
        End Get
        Set
            m_SchoolA = Value
        End Set
    End Property
    Private m_SchoolA As Double
    Public Property SchoolB() As Double
        Get
            Return m_SchoolB
        End Get
        Set
            m_SchoolB = Value
        End Set
    End Property
    Private m_SchoolB As Double
    Public Property SchoolC() As Double
        Get
            Return m_SchoolC
        End Get
        Set
            m_SchoolC = Value
        End Set
    End Property
    Private m_SchoolC As Double
End Class
```

- **DataCreator.cs**

```
namespace BoxWhiskers
{
    class DataCreator
    {
        public static List<ClassScore> CreateSchoolScoreData()
        {
            var result = new List<ClassScore>();
            result.Add(new ClassScore() { ClassName = "English", SchoolA = 46, SchoolB = 53, SchoolC = 66 });
            result.Add(new ClassScore() { ClassName = "Physics", SchoolA = 61, SchoolB = 55, SchoolC = 65 });
            result.Add(new ClassScore() { ClassName = "English", SchoolA = 58, SchoolB = 56, SchoolC = 67 });
            result.Add(new ClassScore() { ClassName = "Math", SchoolA = 58, SchoolB = 51, SchoolC = 64 });
            result.Add(new ClassScore() { ClassName = "English", SchoolA = 63, SchoolB = 53, SchoolC = 45 });
            result.Add(new ClassScore() { ClassName = "English", SchoolA = 63, SchoolB = 50, SchoolC = 65 });
            result.Add(new ClassScore() { ClassName = "Math", SchoolA = 60, SchoolB = 45, SchoolC = 67 });
            result.Add(new ClassScore() { ClassName = "Math", SchoolA = 62, SchoolB = 53, SchoolC = 66 });
            result.Add(new ClassScore() { ClassName = "Physics", SchoolA = 63, SchoolB = 54, SchoolC = 64 });
            result.Add(new ClassScore() { ClassName = "English", SchoolA = 63, SchoolB = 52, SchoolC = 67 });
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
            result.Add(new ClassScore() { ClassName = "Physics", SchoolA = 69, SchoolB = 66, SchoolC = 71 });
            result.Add(new ClassScore() { ClassName = "Physics", SchoolA = 48, SchoolB = 67, SchoolC = 50 });
            result.Add(new ClassScore() { ClassName = "Physics", SchoolA = 54, SchoolB = 50, SchoolC = 56 });
            result.Add(new ClassScore() { ClassName = "Physics", SchoolA = 60, SchoolB = 56, SchoolC = 64 });
            result.Add(new ClassScore() { ClassName = "Math", SchoolA = 71, SchoolB = 65, SchoolC = 50 });
            result.Add(new ClassScore() { ClassName = "Math", SchoolA = 48, SchoolB = 70, SchoolC = 72 });
            result.Add(new ClassScore() { ClassName = "Math", SchoolA = 53, SchoolB = 40, SchoolC = 80 });
            result.Add(new ClassScore() { ClassName = "Math", SchoolA = 60, SchoolB = 56, SchoolC = 67 });
            result.Add(new ClassScore() { ClassName = "Math", SchoolA = 61, SchoolB = 56, SchoolC = 45 });
            result.Add(new ClassScore() { ClassName = "English", SchoolA = 63, SchoolB = 58, SchoolC = 64 });
            result.Add(new ClassScore() { ClassName = "Physics", SchoolA = 59, SchoolB = 54, SchoolC = 65 });

            return result;
        }
    }

    public class ClassScore
    {
        public string ClassName { get; set; }
        public double SchoolA { get; set; }
        public double SchoolB { get; set; }
        public double SchoolC { get; set; }
    }
```

- **Visual Basic**

```
Partial Public Class MainWindow
    Private _data As List(Of ClassScore) = Nothing
    Public Sub New()
        InitializeComponent()

        ' show mean lines
        boxWhiskerA.ShowMeanLine = True
        boxWhiskerB.ShowMeanLine = True
        boxWhiskerC.ShowMeanLine = True

        ' show inner points
        boxWhiskerA.ShowInnerPoints = True
        boxWhiskerB.ShowInnerPoints = True
        boxWhiskerC.ShowInnerPoints = True

        ' show outliers
        boxWhiskerA.ShowOutliers = True
        boxWhiskerB.ShowOutliers = True
        boxWhiskerC.ShowOutliers = True

        ' show mean marks
        boxWhiskerA.ShowMeanMarks = True
        boxWhiskerB.ShowMeanMarks = True
        boxWhiskerC.ShowMeanMarks = True

        ' specify quartile calculation
        boxWhiskerA.QuartileCalculation = QuartileCalculation.InclusiveMedian
        boxWhiskerB.QuartileCalculation = QuartileCalculation.InclusiveMedian
        boxWhiskerC.QuartileCalculation = QuartileCalculation.InclusiveMedian
    End Sub

End Class

Public ReadOnly Property Data() As List(Of ClassScore)
    Get
        If _data Is Nothing Then
            _data = DataCreator.CreateSchoolScoreData()
        End If

        Return _data
    End Get
End Property
```

- **C#**

```
namespace BoxWhiskers
{
    public partial class BoxWhisker : UserControl
    {
        private List<ClassScore> _data = null;
        public BoxWhisker()
        {
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
            InitializeComponent();

            // show mean lines
            boxWhiskerA.ShowMeanLine = true;
            boxWhiskerB.ShowMeanLine = true;
            boxWhiskerC.ShowMeanLine = true;

            // show inner points
            boxWhiskerA.ShowInnerPoints = true;
            boxWhiskerB.ShowInnerPoints = true;
            boxWhiskerC.ShowInnerPoints = true;

            // show outliers
            boxWhiskerA.ShowOutliers = true;
            boxWhiskerB.ShowOutliers = true;
            boxWhiskerC.ShowOutliers = true;

            // show mean marks
            boxWhiskerA.ShowMeanMarks = true;
            boxWhiskerB.ShowMeanMarks = true;
            boxWhiskerC.ShowMeanMarks = true;

            // specify quartile calculation
            boxWhiskerA.QuartileCalculation = QuartileCalculation.InclusiveMedian;
            boxWhiskerB.QuartileCalculation = QuartileCalculation.InclusiveMedian;
            boxWhiskerC.QuartileCalculation = QuartileCalculation.InclusiveMedian;
        }
        public List<ClassScore> Data
        {
            get
            {
                if (_data == null)
                {
                    _data = DataCreator.CreateSchoolScoreData();
                }

                return _data;
            }
        }
    }
}
```

Error Bar

Error Bar series allows you to indicate variability of data or uncertainty in values. It enables you to display standard deviations and a range of error in variable data using error bars. Generally, results of scientific studies or experimental sciences use error bars in charts to depict variations in data from original values.

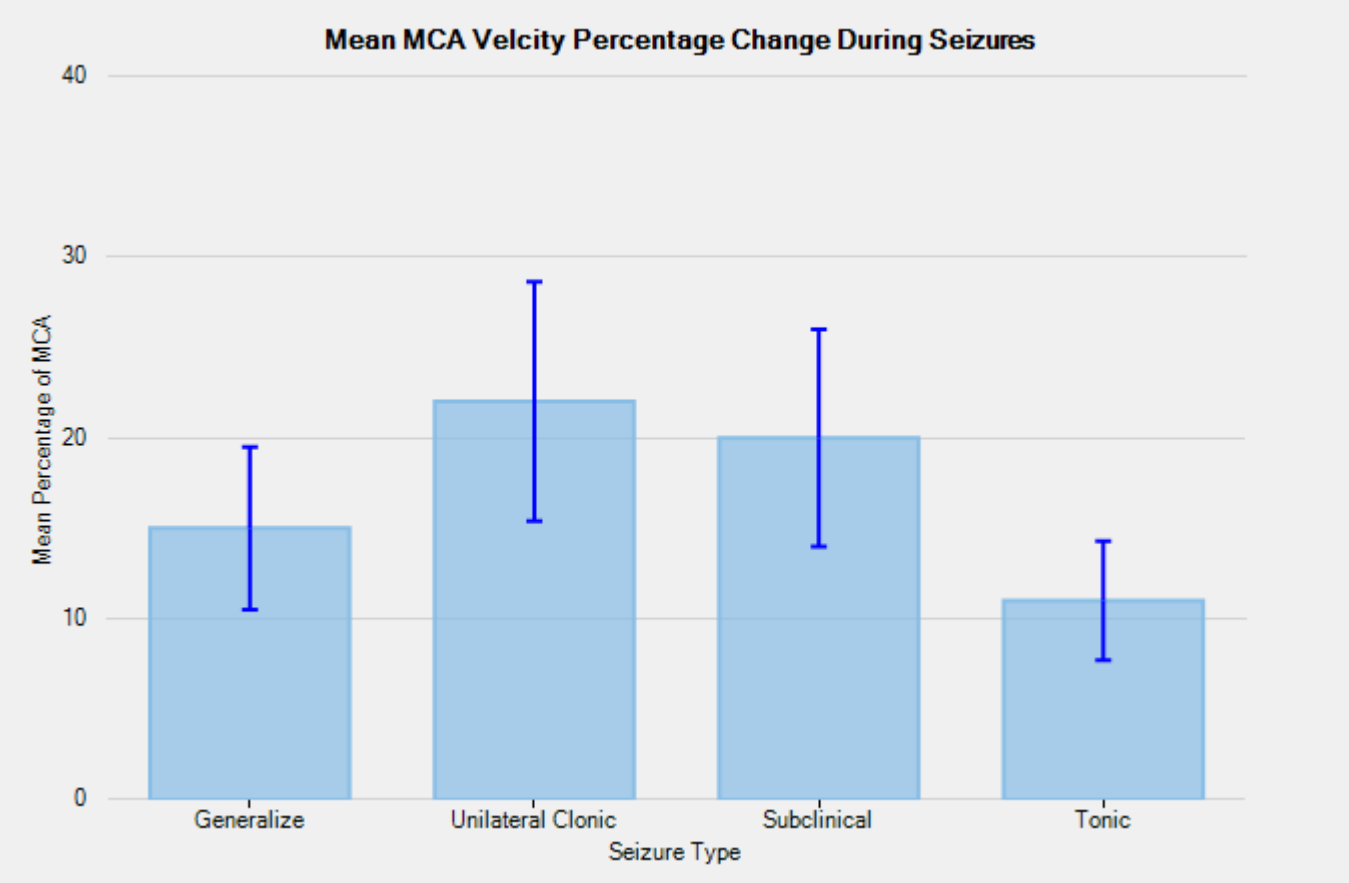
FlexChart lets you use Error Bar series in different chart types including Area, Column, Line, LineSymbols, Scatter, Spline, SplineArea, and SplineSymbols.

Error Bar series in FlexChart offers several features, as follows:

- **Error amount**: Set up error bars on all data points using different ways, such as a fixed value, percentage, standard error, or standard deviation. In addition, it is possible to set a custom value to show a precise error amount, if required. To display error bars in any of these ways, set the ErrorAmount property from the ErrorAmount enumeration.
- **Direction**: Show error bars in the Plus, the Minus, or even both directions by setting the Direction property from the ErrorBarDirection enumeration.
- **End style**: Display error bars with or without caps by setting the EndStyle property from the ErrorBarEndStyle enumeration.
- **Bar style**: Customize the appearance of error bars using the ErrorBarStyle property.

The following image displays Plus and Minus error amounts in the mean MCA (Middle Cerebral Artery) velocity data for different seizure types observed in children.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



The following code uses mean percentage values of MCA velocity during different kinds of seizures in children. The codes shows how to implement ErrorBar series in FlexChart.

- **DataCreator.vb**

```
Class DataCreator
    Public Shared Function CreateData() As List(Of DataItem)
        Dim data = New List(Of DataItem)()
        data.Add(New DataItem("Generalize", 15))
        data.Add(New DataItem("Unilateral Clonic", 22))
        data.Add(New DataItem("Subclinical", 20))
        data.Add(New DataItem("Tonic", 11))
        Return data
    End Function
End Class
Public Class DataItem
    Public Sub New(seizuretype__1 As String, meanmca__2 As Integer)
        SeizureType = seizuretype__1
        MeanMCA = meanmca__2
    End Sub

    Public Property SeizureType() As String
        Get
            Return m_SeizureType
        End Get
        Set
            m_SeizureType = Value
        End Set
    End Property
    Private m_SeizureType As String
    Public Property MeanMCA() As Integer
        Get
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
            Return m_MeanMCA
        End Get
        Set
            m_MeanMCA = Value
        End Set
    End Property
    Private m_MeanMCA As Integer
End Class
```

- **DataCreator.cs**

```
class DataCreator
{
    public static List<DataItem> CreateData()
    {
        var data = new List<DataItem>();
        data.Add(new DataItem("Generalize", 15));
        data.Add(new DataItem("Unilateral Clonic", 22));
        data.Add(new DataItem("Subclinical", 20));
        data.Add(new DataItem("Tonic", 11));
        return data;
    }
}
public class DataItem
{
    public DataItem(string seizuretype, int meanmca)
    {
        SeizureType = seizuretype;
        MeanMCA = meanmca;
    }

    public string SeizureType { get; set; }
    public int MeanMCA { get; set; }
}
```

- **Visual Basic**

```
Partial Public Class MainWindow
    Inherits Window
    Private _data As List(Of DataItem)
    Public Sub New()
        InitializeComponent()

        ' clear data series collection
        flexChart.Series.Clear()

        ' create ErrorBar series
        Dim errorBar As New C1.WPF.Chart.ErrorBar()

        ' add the series to the data series collection
        flexChart.Series.Add(errorBar)

        ' bind X-axis and Y-axis
        flexChart.BindingX = "SeizureType"
        errorBar.Binding = "MeanMCA"

        ' specify error amount of the series
        errorBar.ErrorAmount = C1.Chart.ErrorAmount.Percentage

        ' specify the direction of the error
        errorBar.Direction = C1.Chart.ErrorBarDirection.Both
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
        ' specify the error value of the series
        errorBar.ErrorValue = 0.3

        ' style the ErrorBar series
        errorBar.EndStyle = C1.Chart.ErrorBarEndStyle.Cap
    End Sub
    Public ReadOnly Property Data() As List(Of DataItem)
        Get
            If _data Is Nothing Then
                _data = DataCreator.CreateData()
            End If

            Return _data
        End Get
    End Property
End Class
```

- **C#**

```
public partial class MainWindow : Window
{
    private List<DataItem> _data;
    public MainWindow()
    {
        InitializeComponent();

        // clear data series collection
        flexChart.Series.Clear();

        // create ErrorBar series
        C1.WPF.Chart.ErrorBar errorBar = new C1.WPF.Chart.ErrorBar();

        // add the series to the data series collection
        flexChart.Series.Add(errorBar);

        // bind X-axis and Y-axis
        flexChart.BindingX = "SeizureType";
        errorBar.Binding = "MeanMCA";

        // specify error amount of the series
        errorBar.ErrorAmount = C1.Chart.ErrorAmount.Percentage;

        // specify the direction of the error
        errorBar.Direction = C1.Chart.ErrorBarDirection.Both;

        // specify the error value of the series
        errorBar.ErrorValue = .3;

        // style the ErrorBar series
        errorBar.EndStyle = C1.Chart.ErrorBarEndStyle.Cap;

    }
    public List<DataItem> Data
    {
        get
        {
            if (_data == null)
            {
                _data = DataCreator.CreateData();
            }

            return _data;
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
            }
        }
}
```

Waterfall Series

Waterfall series allows you to understand the cumulative effect of sequential positive or negative values. It is useful to understand the effect of a series of positive and negative values on an initial value. The series depicts color coded columns to easily distinguish positive values from negative values. Generally initial and final values are depicted by total columns, while intermediate values are represented by floating columns. It is recommended to use Waterfall series when there is a column of category text and a mix of positive and negative values. Such cases are mostly found in quantitative analysis like inventory analysis or performance analysis, where the chart shows the gradual transition in the quantitative value of an entity subjected to increment or decrement.

FlexChart provides features that can be implemented and customized for enhanced data visualization through Waterfall series.

- **Connector lines**: Connector lines are the lines that connect color coded columns to display the flow of data in the chart. To show connector lines, set the ConnectorLines property of the WaterFall class to True.
- **Connector lines customization**: Once displayed, connector lines can be customized using the ConnectorLineStyle property that allows you to access styling properties of the ChartStyle class.
- **Columns customization**: To clearly differentiate positive values from negative values or total, you can apply various styles to the columns showing these values. For that, you can use various properties, such as RisingStyle, FallingStyle, TotalStyle, or StartStyle provided by the Waterfall class.

The following image displays Waterfall series displaying the cumulative effect of sequential positive and negative values.

To use the Waterfall series in FlexChart, create an instance of the Waterfall class, which inherits the Series class, and add the created instance to the FlexChart Series collection using the Series property provided by the C1FlexChart class.

The following code snippet illustrates how to set various properties while working with Waterfall series in FlexChart. The code snippet first creates a class DataCreator to generate data for the chart, and then binds the series to the data source.

- **Visual Basic**

```
Class DataCreator
    Public Shared Function CreateData() As List(Of DataItem)
        Dim data = New List(Of DataItem)()
        data.Add(New DataItem("製品の収入", 420))
        data.Add(New DataItem("サービスの収入", -180))
        data.Add(New DataItem("固定費", 130))
        data.Add(New DataItem("変動費用", -20))
        Return data
    End Function
End Class

Public Class DataItem
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
    Public Sub New(costs__1 As String, amount__2 As Integer)
        Costs = costs__1
        Amount = amount__2
    End Sub

    Public Property Costs() As String
        Get
            Return m_Costs
        End Get
        Set
            m_Costs = Value
        End Set
    End Property
    Private m_Costs As String
    Public Property Amount() As Integer
        Get
            Return m_Amount
        End Get
        Set
            m_Amount = Value
        End Set
    End Property
    Private m_Amount As Integer
End Class
```

- **C#**

```
using System.Collections.Generic;

namespace Waterfall
{
    class DataCreator
    {
        public static List<DataItem> CreateData()
        {
            var data = new List<DataItem>();
            data.Add(new DataItem("製品の収入", 420));
            data.Add(new DataItem("サービスの収入", -180));
            data.Add(new DataItem("固定費", 130));
            data.Add(new DataItem("変動費用", -20));
            return data;
        }
    }

    public class DataItem
    {
        public DataItem(string costs, int amount)
        {
            Costs = costs;
            Amount = amount;
        }

        public string Costs { get; set; }
        public int Amount { get; set; }
    }
}
```

Here is the code snippet for binding the FlexChart to the data source.

XAML

```
<Window
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:Waterfall"
        xmlns:c1="http://schemas.componentone.com/winfx/2006/xaml" x:Class="Waterfall.MainWindow"
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
        mc:Ignorable="d"
        DataContext="{Binding RelativeSource={RelativeSource Mode=Self}}"
        Title="MainWindow" Height="528.558" Width="712.292">

        <Grid>

            <c1:C1FlexChart x:Name="flexChart"
                            BindingX="Costs"
                            ItemsSource="{Binding DataContext.Data}"
                            Margin="55,161,51,28">
                <c1:C1FlexChart.Series>
                    <c1:Waterfall Binding="Amount"
                                  ConnectorLines="True"
                                  ShowTotal="True"
                                  ShowIntermediateTotal="True">
                        <c1:Waterfall.StartStyle>
                            <c1:ChartStyle Fill="#7dc7ed" />
                        </c1:Waterfall.StartStyle>
                        <c1:Waterfall.FallingStyle>
                            <c1:ChartStyle Fill="#dd2714" />
                        </c1:Waterfall.FallingStyle>
                        <c1:Waterfall.RisingStyle>
                            <c1:ChartStyle Fill="#0f9d58"
                                           Stroke="#0f9d58" />
                        </c1:Waterfall.RisingStyle>
                        <c1:Waterfall.IntermediateTotalStyle>
                            <c1:ChartStyle Fill="#7dc7ed" />
                        </c1:Waterfall.IntermediateTotalStyle>
                        <c1:Waterfall.TotalStyle>
                            <c1:ChartStyle Fill="#7dc7ed" />
                        </c1:Waterfall.TotalStyle>
                        <c1:Waterfall.ConnectorLineStyle>
                            <c1:ChartStyle Stroke="#333"
                                           StrokeDashArray="5,5"/>
                        </c1:Waterfall.ConnectorLineStyle>
                    </c1:Waterfall>
                </c1:C1FlexChart.Series>
                <c1:C1FlexChart.AxisY>
                    <c1:Axis Min="0"></c1:Axis>
                </c1:C1FlexChart.AxisY>
                </c1:C1FlexChart>

        </Grid>
</Window>
```

Code

C# copyCode

```
// clear the FlexChart series collection
flexChart.Series.Clear();

// create an instance of the Waterfall series
C1.WPF.Chart.Waterfall waterFall = new C1.WPF.Chart.Waterfall();

// add the instance to Series collection
flexChart.Series.Add(waterFall);

// bind the field containing Y values for the series
waterFall.Binding = "Amount";

// bind the field containing X values for the FlexChart
flexChart.BindingX = "Costs";

// set the ConnectorLines property
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
waterFall.ConnectorLines = true;

// set the ShowTotal property
waterFall.ShowTotal = true;
```

VB copyCode

```
' clear the FlexChart series collection
flexChart.Series.Clear()

' create an instance of the Waterfall series
Dim waterFall As New C1.WPF.Chart.Waterfall()

' add the instance to Series collection
flexChart.Series.Add(waterFall)

' bind the field containing Y values for the series
waterFall.Binding = "Amount"

' bind the field containing X values for the FlexChart
flexChart.BindingX = "Costs"

' set the ConnectorLines property
waterFall.ConnectorLines = True

' set the ShowTotal property
waterFall.ShowTotal = True
```

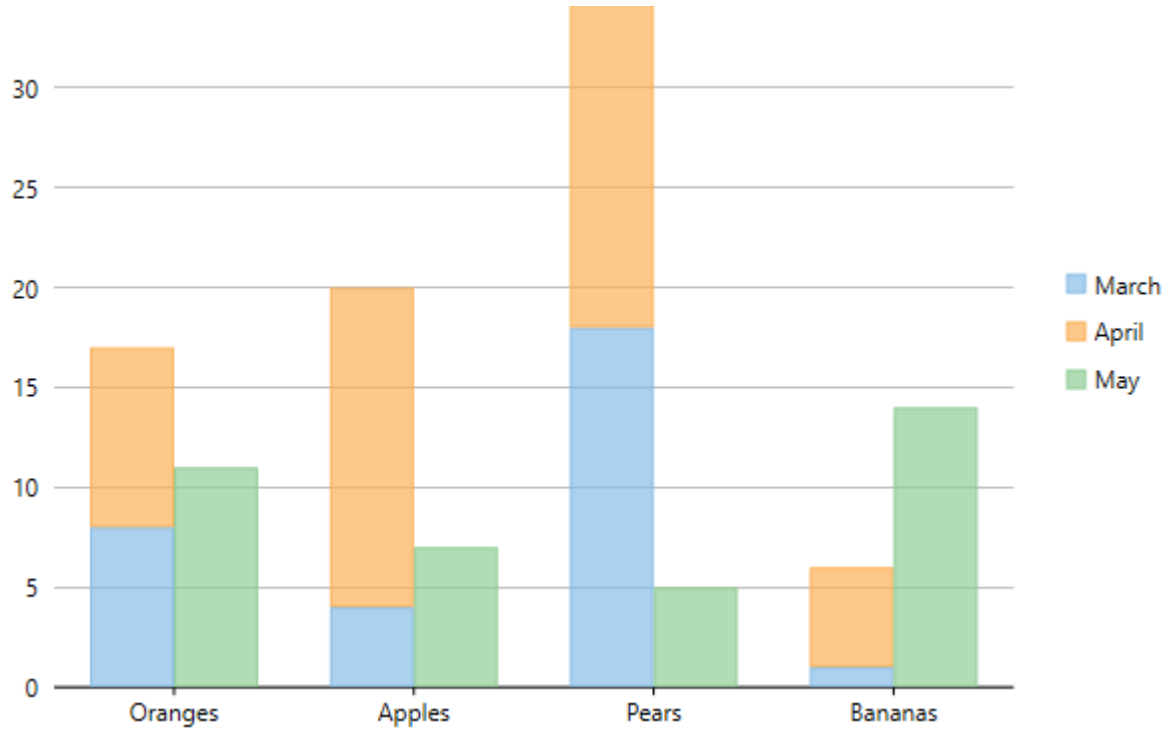
Stacked Groups

FlexChart supports stacking and grouping of data items in column and bar charts. Stacking provides capabilities for stacking data items one on top of the other (in column chart) or side-by-side (in bar chart). Whereas, grouping enables clustering of the stacked data items in bar and column charts.

Stacked groups allow you to compare items across categories in a group. In addition, you can visualize relative difference between items in each group.

The following image displays stacked groups in FlexChart.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



To stack specific series in a specific stacked group, set the index value of that stacked group in the StackingGroup property for the series. Note that Stacked groups in FlexChart are implementable when the Stacking property for FlexChart is set to either **Stacked** or **Stacked100pc**, which specifies how the data values of chart will be stacked.

The following code compares fruit data for three consecutive months and shows how to implement stacked groups in FlexChart.

- **XAML**

```
<Chart:C1FlexChart x:Name="flexChart"
                   Stacking="Stacked"
                   ItemsSource="{Binding DataContext.Data}"
                   BindingX="Fruit">
    <Chart:Series SeriesName="March" Binding="March"
                  StackingGroup="0"/>
    <Chart:Series SeriesName="April" Binding="April"
                  StackingGroup="0"/>
    <Chart:Series SeriesName="May" Binding="May"
                  StackingGroup="1"/>
</Chart:C1FlexChart>
```

FlexChart Data Labels

Data labels are the labels associated with data points to provide additional information about the data points. In other words, these labels can be defined as descriptive texts or values displayed over data points of the series. These labels are primarily used to highlight crucial data points, thereby enhancing the readability of the chart and allowing you to identify data quickly.

FlexChart offers support for highly customizable data labels that enable you to conveniently highlight chart data. And that in turn helps end-users to identify and interpret the chart data more efficiently. When it comes to working with data labels in FlexChart, the DataLabel property allows you to do so. By default, FlexChart does not display data labels; however, you can not only display data labels, but also customize them as per your requirements using various properties of the DataLabel and the DataLabelBase classes.

Below are the sections that describe how you can add data labels to data points and how you can control data labels

Copyright © 2019 GrapeCity, Inc. All rights reserved.

in terms of their appearance and the data they display:

- Adding and Positioning Data Labels
- Formatting Data Labels
- Manage Overlapped Data Labels

Adding and Positioning Data Labels

When added to data points in the chart, data labels make it easier to understand the chart data because they display details about individual data points. These labels quickly highlight data that is both relevant and important.

There is a simple method to add data labels to data points while working with FlexChart. You just need to configure the Content property as per the type of entry you want to display in data labels. And you need to set the position of data labels using the Position property to display the data labels in the chart.

The table below lists the pre-defined parameters applicable for data label content customization.

Parameter	Description
x	Shows the X value of the data point.
y	Shows the Y value of the data point.
value	Shows the Y value of the data point.
name	Shows the X value of the data point.
seriesName	Shows the name of the series.
pointIndex	Shows the index of the data point.
P	Shows the percentage share with respect to the parent slice in Sunburst.
p	Shows the percentage share with respect to the whole chart in Sunburst.

See the following code snippet for reference.

Here is the code that demonstrates a fully-functional program implementing data binding:

XAML

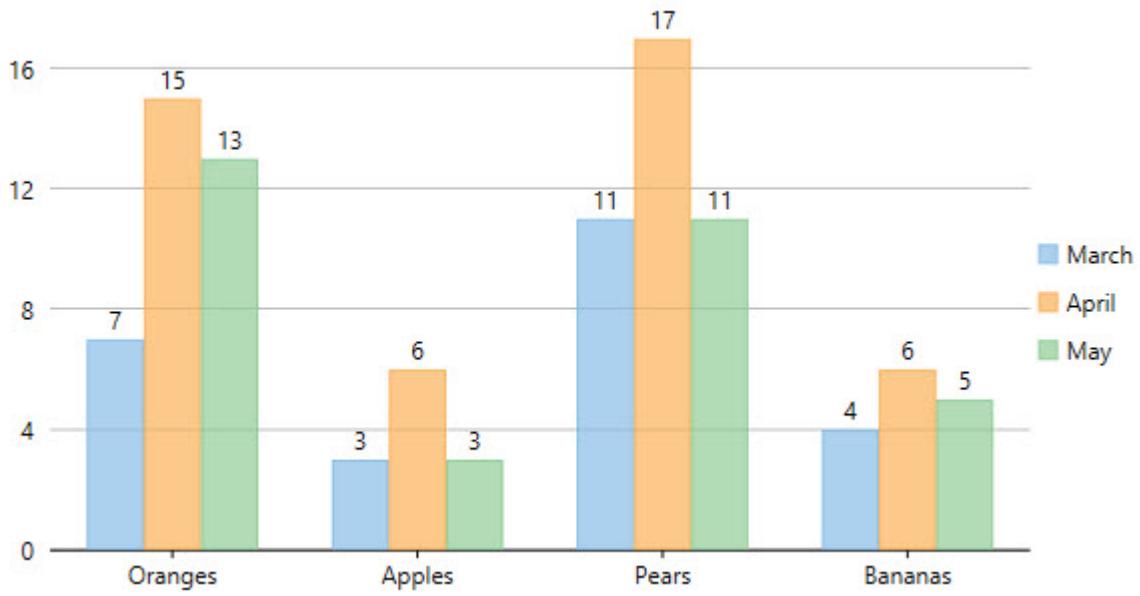
```
<Chart:C1FlexChart.DataLabel>
    <Chart:DataLabel Content="{}{y}" Position="Top"/>
</Chart:C1FlexChart.DataLabel>
```

Code

C# copyCode

```
flexChart.DataLabel.Content = "{y}";
flexChart.DataLabel.Position = LabelPosition.Top;
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.



Depending upon the chart type, you can select from different positioning options to position data labels perfectly in the chart. The Position property accepts the following values from the LabelPosition enumeration:

Property	Description
Top	Sets the labels above the data points.
Bottom	Sets the labels below the data points.
Left	Sets the labels to the left of the data points.
Right	Sets the labels to the right of the data points.
Center	Sets the labels centered on the data points.
None	Hides the labels.

Using the Content property, you can customize the content of the data labels to further include series names, index values, or names of data points.

The following code shows how to set the Content property to include series names and data point values in data labels.

XAML

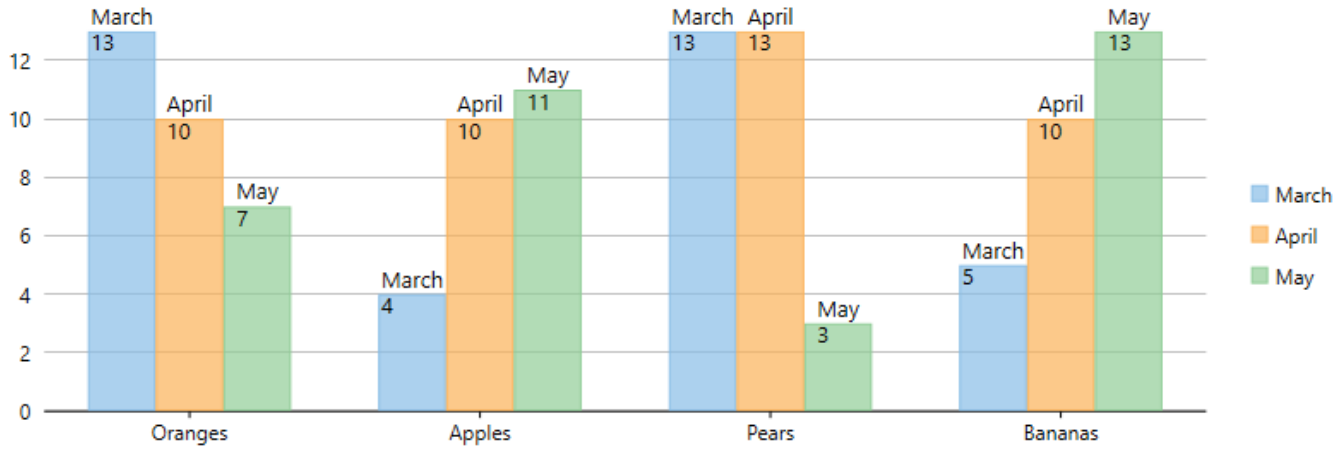
```
<c1:C1FlexChart.DataLabel>
    <c1:DataLabel Content="{}{seriesName}&#x0a;{y}"/>
</c1:C1FlexChart.DataLabel>
```

Code

C# copyCode

```
// set the Content property
flexChart.DataLabel.Content = "{seriesName}\n{value}";
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.



Formatting Data Labels

FlexChart provides a number of options to format data labels the way you want. You can set and style borders of data labels, connect them with their corresponding data points, and customize the way data labels appear.

Setting and Styling Borders of Data Labels

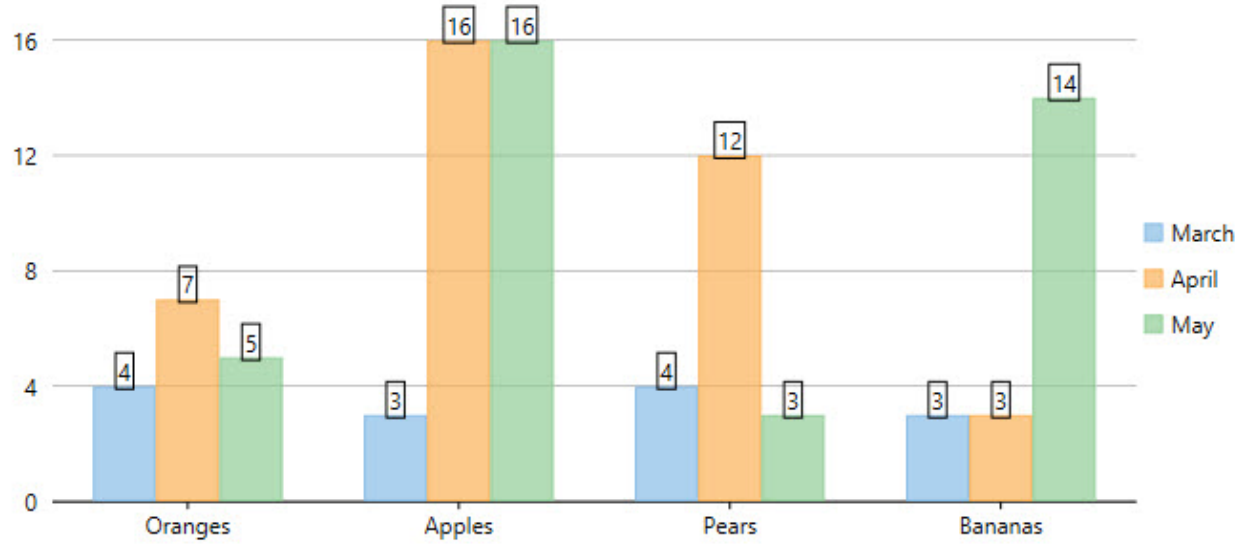
Borders add an extra appeal to data labels and make them more highlighted. This comes in handy to seamlessly highlight really crucial data in the chart, so that end users can focus on what is important.

In FlexChart, borders can be enabled and customized by using different properties, such as Border and **BorderStyle**.

Here is the code snippet illustrating the setting and the customization of borders.

- **C#**

```
flexChart.DataLabel.Border = true;
```



Connecting Data Labels to Data Points

If you have placed data labels away from their corresponding data points, you can connect them using leader lines. A leader line is a line that connects a data label to its data point. Leader lines are beneficial to use, especially when you need to display a visual connection between data labels and their associated data points.

FlexChart does not display leader lines by default when you add data labels; nevertheless, you can enable leader lines and even set their appropriate length to create better visual connections for data labels in the chart. To enable leader lines, you

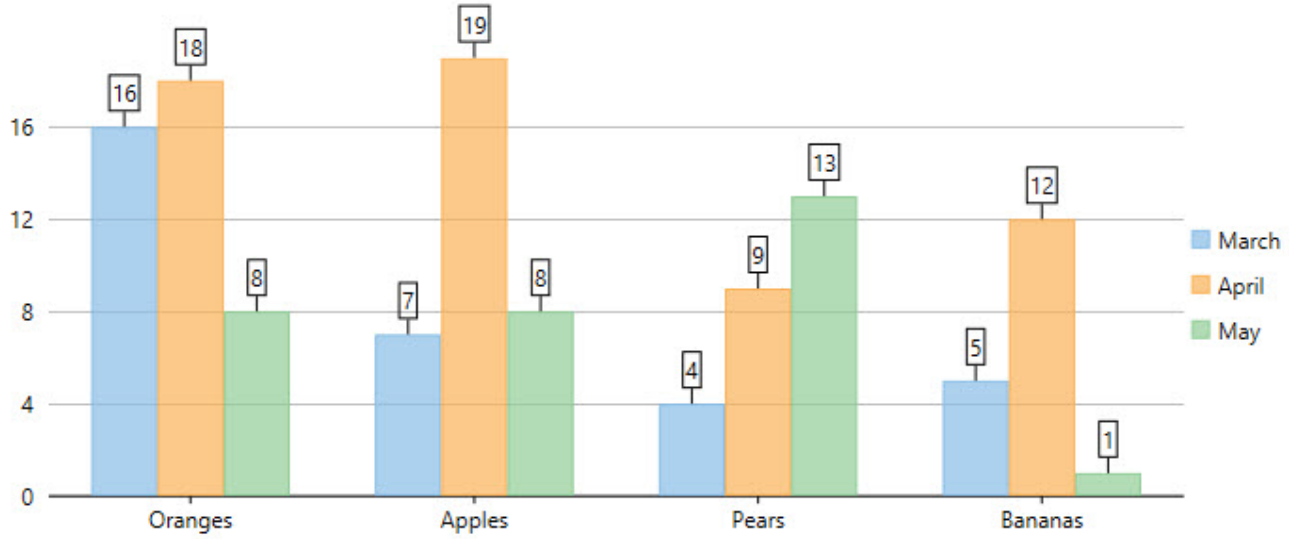
Copyright © 2019 GrapeCity, Inc. All rights reserved.

need to use the ConnectingLine property. And to set the distance between data labels and their data points, you need to use the Offset property.

The following code snippet sets both the properties.

- **C#**

```
flexChart.DataLabel.ConnectingLine = true;
flexChart.DataLabel.Offset = 10;
```



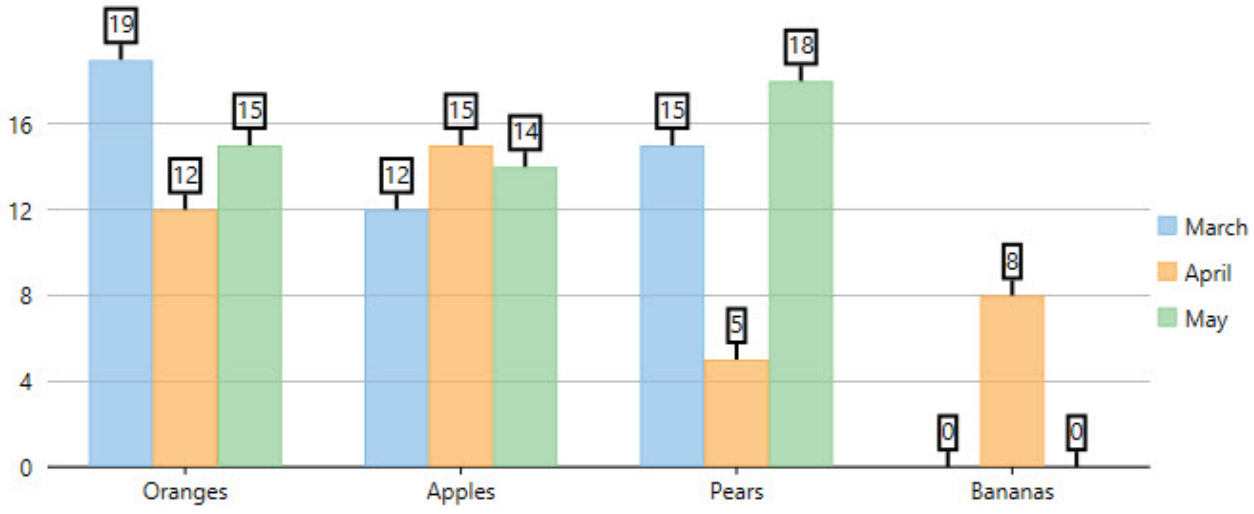
Changing the Appearance of Data Labels

You can make data visualization powerful and appealing by changing the way data labels appear in the chart. FlexChart contains various styling options, which you can use to enhance the clarity and look of data labels. You can use the Style property to change the appearance of data labels.

See the following code snippet for reference.

- **C#**

```
flexChart.DataLabel.Style.FontFamily = new System.Windows.Media.FontFamily("GenericSerif");
flexChart.DataLabel.Style.FontSize = 13;
flexChart.DataLabel.Style.StrokeThickness = 2;
```



Copyright © 2019 GrapeCity, Inc. All rights reserved.

Manage Overlapped Data Labels

A common issue pertaining to charts is the overlapping of data labels that represent data points. In most cases, overlapping occurs due to long text in data labels or large numbers of data points.

In case of overlapped data labels in FlexChart, it provides the following ways to manage the overlapping.

- **Auto Arrangement of Data Labels**
- **Hide Overlapped Labels**
- **Control Appearance of Overlapped Labels**
- **Rotate Data Labels**
- **Trim or Wrap Data Labels**

Auto Arrangement of Data Labels

The easiest way to handle overlapping of data labels is to set the FlexChart to position the data labels automatically. For automatic positioning of data labels, you can set the Position property to Auto. Moreover, you can also set the MaxAutoLabels property to set the maximum number of labels that can be positioned automatically.

When the Position property is set to Auto, the number of created data labels is limited by MaxAutoLabels property which is 100 by default. You can increase the value of MaxAutoLabels property if necessary, but it may slow down the chart rendering since the label positioning algorithm becomes expensive in terms of performance when number of labels is large.

This approach may not provide an optimal layout when working with large data set and when there is not enough space for all data labels. In this case, it's recommended to reduce the number of data labels. For example, create a series with limited number of data points that should have labels, that is, chose to hide the labels at the individual series level.

C#

```
// Set Position and MaxAutoLabels property
flexChart.DataLabel.Position = LabelPosition.Auto;
flexChart.DataLabel.MaxAutoLabels = 150;
```

Hide Overlapped Labels

In case of overlapped data labels in FlexChart, you can use the Overlapping property provided by DataLabel class. This approach is helpful when the developer wants to completely hide or show the overlapped data labels.

C#

```
// Set Overlapping property
flexChart.DataLabel.Overlapping = LabelOverlapping.Hide;
```

The Overlapping property accepts the following values in the LabelOverlapping enumeration.

Enumeration	Description
Hide	Hide overlapped data labels.
Show	Show overlapped data labels.

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Control Appearance of Overlapped Labels

Furthermore, you can use the OverlappingOptions property to specify additional label overlapping options that will help the user to effectively manage overlapping of data labels.

C#

```
// Set OverlappingOptions property
flexChart.DataLabel.OverlappingOptions =
LabelOverlappingOptions.OutsidePlotArea;
```

The OverlappingOptions property accepts the following values in the LabelOverlappingOptions enumeration.

Enumeration	Description
None	No overlapping is allowed.
OutsidePlotArea	Allow labels outside plot area.
OverlapDataPoints	Allow overlapping with data points.

Rotate Data Labels

Another option to manage overlapping of data labels in FlexChart is to use the Angle property. The Angle property enables the user to set a specific rotation angle for data labels.

C#

```
// Set the Angle property
flexChart.DataLabel.Angle = 50;
```

Trim or Wrap Data Labels

To manage the content displayed in the data labels, in case of overlapping, you can either trim the data labels or wrap the data labels using ContentOptions property. Managing of data labels using the ContenOptions property is dependent on MaxWidth and MaxLines property.

The MaxWidth property allows you to set the maximum width of a data label. In case the width of data label text exceeds the specified width, then you can either trim the data labels or wrap the data labels using the ContentOptions property.

The MaxLines property allows you to set the maximum number of lines in data label. This property helps you to limit the wrapped text to grow vertically. In case the wrapped text does not fit within the specified MaxWidth and MaxLines property values, then the last line gets trimmed with an ellipses(...).

C#

```
// Set MaxWidth property
flexChart.DataLabel.MaxWidth = 25;      // Set ContentOptions property
flexChart.DataLabel.ContentOptions = ContentOptions.Trim;
```

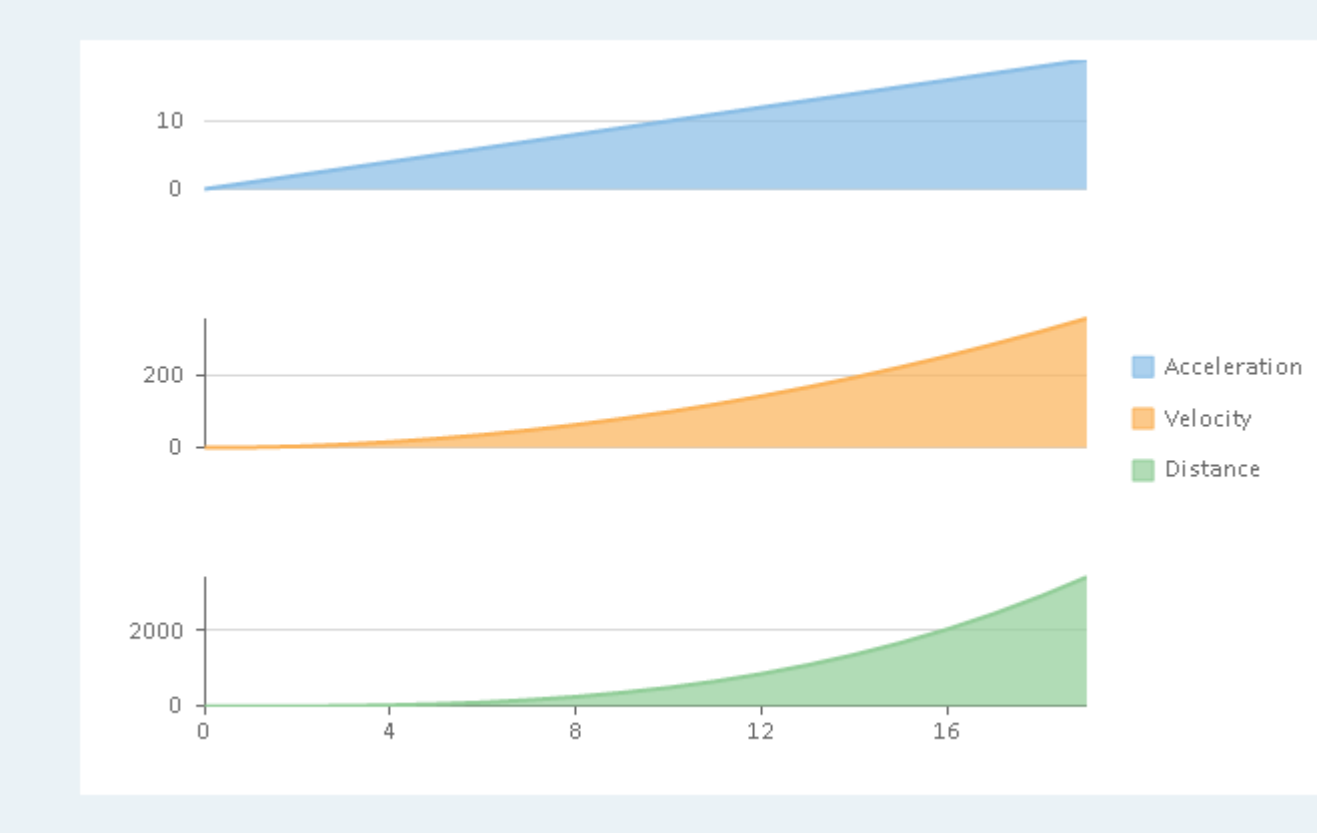
Copyright © 2019 GrapeCity, Inc. All rights reserved.

Multiple Plot Areas

Multiple plot areas allow you to increase the visibility of data by displaying each series in a separate plot area across one axis, keeping the other axis fixed.

FlexChart enables you to create multiple plot areas for different series within the same chart area. In FlexChart, create different plot areas and add them to the C1FlexChart.PlotAreas collection. In addition, you can customize the plot areas in terms of row index, column index, height, and width.

The following image displays multiple plot areas showing data for one series each in FlexChart.



The following code uses data regarding four metrics, namely, Acceleration, Velocity, Distance, and Time of a vehicle. The code demonstrates how to implement multiple plot areas in FlexChart.

- **Visual Basic**

```
' create and add multiple plot areas
flexChart.PlotAreas.Add(New PlotArea() With {
    .PlotAreaName = "plot1",
    .Row = 0
})
flexChart.PlotAreas.Add(New PlotArea() With {
    .PlotAreaName = "plot2",
    .Row = 2
})
flexChart.PlotAreas.Add(New PlotArea() With {
    .PlotAreaName = "plot3",
    .Row = 4
})

' specify the chart type
flexChart.ChartType = C1.Chart.ChartType.Area
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
' create, add, and bind series
flexChart.Series.Add(New Series() With {
    .SeriesName = "Acceleration",
    .Binding = "Acceleration"
})

flexChart.Series.Add(New Series() With {
    .SeriesName = "Velocity",
    .Binding = "Velocity",
    .AxisY = New Axis() With {
        .Position = C1.Chart.Position.Left,
        .MajorGrid = True,
        .PlotAreaName = "plot2"
    }
})

flexChart.Series.Add(New Series() With {
    .SeriesName = "Distance",
    .Binding = "Distance",
    .AxisY = New Axis() With {
        .Position = C1.Chart.Position.Left,
        .MajorGrid = True,
        .PlotAreaName = "plot3"
    }
})
```

- **C#**

```
// create and add multiple plot areas
flexChart.PlotAreas.Add(new PlotArea { PlotAreaName = "plot1", Row = 0 });
flexChart.PlotAreas.Add(new PlotArea { PlotAreaName = "plot2", Row = 2 });
flexChart.PlotAreas.Add(new PlotArea { PlotAreaName = "plot3", Row = 4 });

// specify the chart type
flexChart.ChartType = C1.Chart.ChartType.Area;

// create, add, and bind series
flexChart.Series.Add(new Series()
{
    SeriesName = "Acceleration",
    Binding = "Acceleration",

});

flexChart.Series.Add(new Series()
{
    SeriesName = "Velocity",
    Binding = "Velocity",
    AxisY = new Axis()
    {
        Position = C1.Chart.Position.Left,
        MajorGrid = true,
        PlotAreaName = "plot2"
    },

});

flexChart.Series.Add(new Series()
{
    SeriesName = "Distance",
    Binding = "Distance",
    AxisY = new Axis()
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
    {
        Position = C1.Chart.Position.Left,
        MajorGrid = true,
        PlotAreaName = "plot3"
    }
});
```

Trend Lines

Trend lines are an important tool for analyzing data. Trend line indicates the general rate of increase or decrease of Y data over X data in a chart. A common scenario is measuring the rate change of sales price over time. FlexChart control supports trend lines through a built-in TrendLine class for ease of use during implementation. Trend lines are most commonly used in Line, Column, Bar, or Scatter charts.

FlexChart supports the following regression and non-regression trend lines.

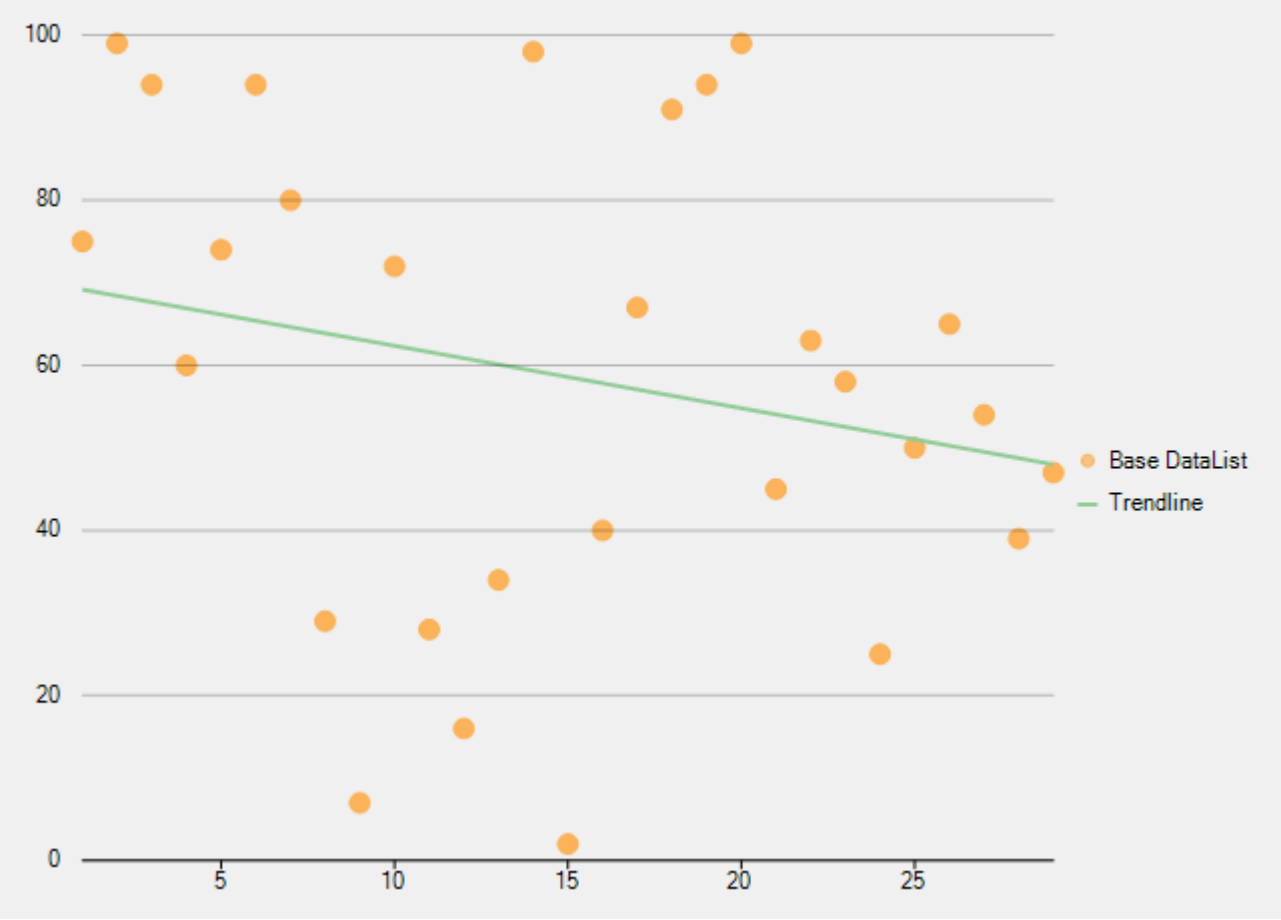
TrendLine.FitType	Description
Linear	A linear trend line is the straight line that most closely approximates the data in the chart. The data is linear, if the data pattern resembles a line. Equation - Y(x) = C0 + C1*x
Polynomial	Polynomial trend lines are curved lines that are used with fluctuating data. They are useful for analyzing gains or losses over a large data set. When using a polynomial trend line, it is important to also set the Order of the line, which can be determined by the number of fluctuations in the data. Equation - Y(x) = C0 + C1*x + C2*x2 + : + Cn-1*xn-1
Logarithmic	Logarithmic trend line is a best-fit curved line that is most useful when the rate of change in the data increases or decreases quickly and then levels out. A logarithmic trend line can use negative and/or positive values. Equation - Y(x) = C0 * ln(C1*x)
Power	Power trend line is a curved line that is best used with data sets that compare measurements that increase at a specific rate — for example, the acceleration of a race car at one-second intervals. You cannot create a power trend line if your data contains zero or negative values. Equation - Y(x) = C0 * pow(x, C1)
Exponent	Exponential trend line is a curved line that is most useful when data values rise or fall at increasingly higher rates. You cannot create an exponential trend line if your data contains zero or negative values. Equation - Y(x) = C0 * exp(C1*x)
Fourier	Fourier trend line identifies patterns or cycles in a series data set. It removes the effects of trends or other complicating factors from the data set, thereby providing a good estimate of the direction that the data under analysis will take in the future. Equation - Y(x) = C0 + C1 * cos(x) + C2 * sin(x) + C3 * cos(2*x) + C4 * sin(2*x) + ...
MinX	The minimum X-value on the chart.
MinY	The minimum Y-value on the chart.
MaxX	The maximum X-value on the chart.

Copyright © 2019 GrapeCity, Inc. All rights reserved.

MaxY	The maximum Y-value on the chart.
AverageX	The average X-value on the chart.
AverageY	The average Y-value on the chart.

To implement trend line in FlexChart, use the TrendLine class that inherits the Series class. To begin with, create an instance of TrendLine class, and then use the TrendLine object to specify its properties. Once, the associated properties are set, add the trend line to the FlexChart using Series.Add method.

The following image shows how FlexChart appears after adding a linear trend line.



XAML

```
<c1:C1FlexChart x:Name="flexChart" Background="White" ChartType="Scatter"
ItemsSource="{Binding Data}" ToolTipContent="{}{y}" Grid.Row="1">
            <c1:C1FlexChart.AxisY>
                <c1:Axis Min="0" Max="100" AxisLine="False" MajorGrid="True"
MajorTickMarks="None" />
            </c1:C1FlexChart.AxisY>
            <c1:Series SeriesName="Base Data" BindingX="X" Binding="Y"/>
            <c1:TrendLine SeriesName="Trend Line" x:Name="trendLine" Binding="Y"
BindingX="X" Order="4"/>
        </c1:C1FlexChart>
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Code

HTML

```
public partial class Trendline
    {
        ObservableCollection<DataItem> dataList = new ObservableCollection<DataItem>
();
        public Form1()
        {
            InitializeComponent();
            var rnt = new Random();
            for (int i = 1; i < 30; i++)
            {
                dataList.Add(new DataItem() { X = i, Y = rnt.Next(100) });
            }
        }
        public class DataItem
        {
            int _y;
            public int X { get; set; }

            public int Y
            {
                get { return _y; }
                set
                {
                    if (value == _y) return;
                    _y = value;
                }
            }}
        }
    }
```

Export

Export to Image

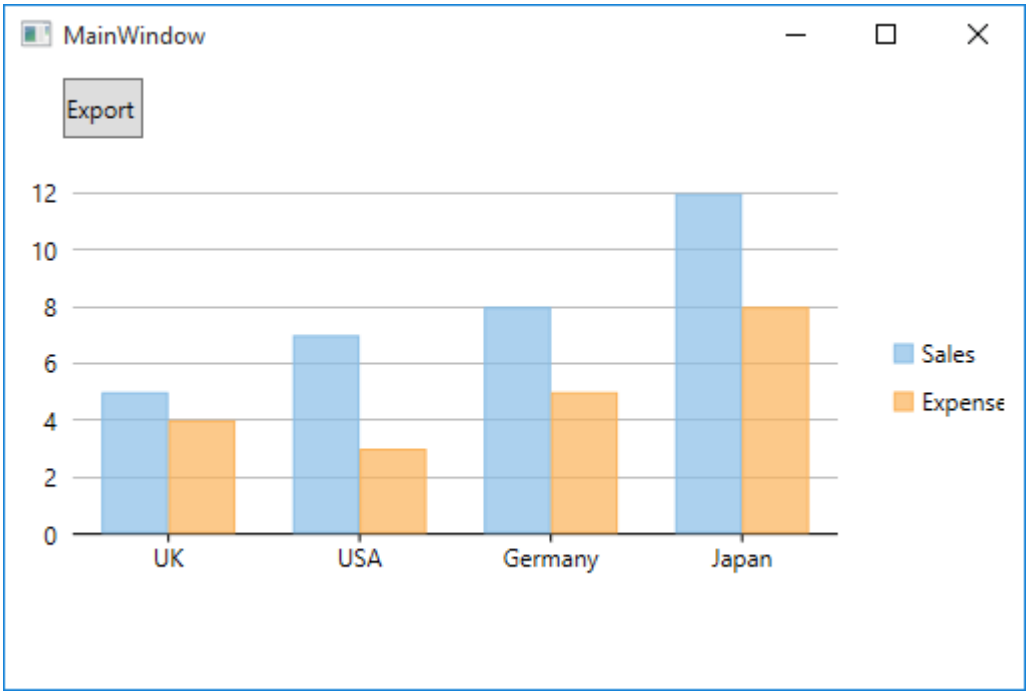
FlexChart for WPF allows you to export the chart to multiple image formats. The supported formats are **PNG**, **JPEG**, **BMP** and **SVG**.

To export a FlexChart to an image format, use **SaveImage** method. The method saves the chart as an image to the specified stream in the given ImageFormat. You can optionally specify the height, width, and back color of the image to be saved.

This topic uses the sample created in QuickStart topic to explain the implementation for exporting a FlexChart to an image on button click event.

The following image shows a chart with a button to be clicked to export chart to a desired image format.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



XAML

MainWindow.xaml

```
<Grid>
        <c1:C1FlexChart x:Name="flexChart"
               BindingX="Country"
               Height="200"
               Width="500"
            ItemsSource="{Binding DataContext.Data}">
            <c1:C1FlexChart.Series>
                <c1:Series SeriesName="Sales"
                       Binding="Sales"/>
                <c1:Series SeriesName="Expenses"
                       Binding="Expenses"/>
            </c1:C1FlexChart.Series>
        </c1:C1FlexChart>
        <Button Content="Export FlexChart" Height="30" Width="40"
Click="OnSaveButtonClick" Margin="0,0,412,269" />
</Grid>
```

Code

MainWindow.xaml.cs

```
public partial class MainWindow : Window
    {
        private List _data;
        public MainWindow()
        {
            InitializeComponent();
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
        }
        public List Data
        {
            get
            {
                if (_data == null)
                {
                    _data = DateCreator.CreateData();
                }

                return _data;
            }
        }

        private void OnSaveButtonClick(object sender, RoutedEventArgs e)
        {
            var dialog = new SaveFileDialog()
            {
                Filter = "PNG|*.png|JPEG |*.jpeg|BMP|*.bmp|SVG|*.svg"
            };
            if (dialog.ShowDialog() == true)
            {
                using (Stream stream = dialog.OpenFile())
                {
                    var extension = dialog.SafeFileName.Split('.')[1];
                    ImageFormat fmt = (ImageFormat)Enum.Parse(typeof(ImageFormat),
extension, true);
                    flexChart.SaveImage(stream, fmt);
                }
            }
        }
    }
```

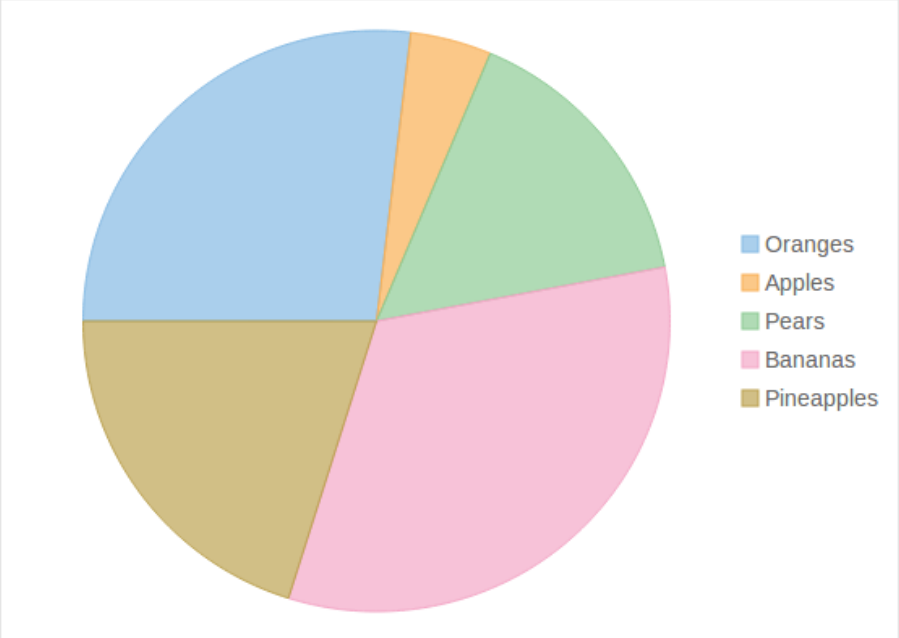
Copyright © 2019 GrapeCity, Inc. All rights reserved.

FlexPie

Pie charts are generally used to represent size of items in a series as a percentage of the entire pie. Ideally, the Pie Chart is to be used when you want to plot only one series comprising non-zero and positive values. And the number of categories are not more than seven.

The **FlexPie** control enables you to create customized pie charts that depict data points as slices of a pie. The arc length of each slice represents the value of that specific slice.

The multi-colored slices make pie charts easy to understand, and usually the value represented by each slice is displayed with the help of labels.



Key Features

- **Header and Footer**: Use simple properties to set a title and footer text.
- **Legend**: Change position of the legend as needed.
- **Selection**: Change the selection mode and customize the selected pie slice appearance.
- **Exploding and Donut Pie Charts**: Use simple properties to convert it into an exploding pie chart or a donut pie chart.
- **Data Labels**: Add, style, format, set the position of data labels and manage the overlapped data labels on the chart.

Quick Start

This quick start is intended to guide you through a step-by-step process of creating a simple FlexPie application and running the same in Visual Studio.

Complete the following steps to see how FlexPie appears on running the application:

- **Step1: Adding FlexPie to the Application**

Copyright © 2019 GrapeCity, Inc. All rights reserved.

- **Step 2: Binding FlexPie to a Data Source**
- **Step 3: Running the Application**

Step 1: Adding FlexPie to the Application

1. Create a **WPF Application** in Visual Studio.
2. Drag and drop the **C1FlexPie** control to the MainWindow.
   The following dlls are automatically added to the application:
   C1.WPF.4.dll
   C1.WPF.DX.4.dll
   C1.WPF.FlexChart.4.dll

   The XAML markup resembles the following code in the **<Grid></Grid>** tags.
   - **XAML**

```
<c1:C1FlexPie x:Name="flexPie"
              Binding="Value"
              BindingName="Name"
              HorizontalAlignment="Left"
              Height="300"
              VerticalAlignment="Top"
              Width="300">
    <c1:C1FlexPie.ItemsSource>
        <c1:FlexPieSliceCollection>
            <c1:FlexPieSlice Name="Slice1" Value="1"/>
            <c1:FlexPieSlice Name="Slice2" Value="2"/>
            <c1:FlexPieSlice Name="Slice3" Value="3"/>
            <c1:FlexPieSlice Name="Slice4" Value="4"/>
        </c1:FlexPieSliceCollection>
    </c1:C1FlexPie.ItemsSource>
</c1:C1FlexPie>
```

Step 2: Binding FlexPie to a Data Source

1. Add a class **DataCreator** and add the following code.
   - **Visual Basic**

```
Public Class DataCreator

    Public Shared Function CreateFruit() As List(Of FruitDataItem)
        Dim fruits = New String() {"Oranges", "Apples", "Pears", "Bananas"}
        Dim count = fruits.Length
        Dim result = New List(Of FruitDataItem)()
        Dim rnd = New Random()
        For i As Object = 0 To count - 1
            result.Add(New FruitDataItem() With {
                .Fruit = fruits(i),
                .March = rnd.[Next](20),
                .April = rnd.[Next](20),
                .May = rnd.[Next](20)
            })
        Next
        Return result
    End Function
End Class

Public Class FruitDataItem
    Public Property Fruit() As String
        Get
            Return m_Fruit
        End Get
        Set
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
            m_Fruit = Value
        End Set
    End Property
    Private m_Fruit As String
    Public Property March() As Double
        Get
            Return m_March
        End Get
        Set
            m_March = Value
        End Set
    End Property
    Private m_March As Double
    Public Property April() As Double
        Get
            Return m_April
        End Get
        Set
            m_April = Value
        End Set
    End Property
    Private m_April As Double
    Public Property May() As Double
        Get
            Return m_May
        End Get
        Set
            m_May = Value
        End Set
    End Property
    Private m_May As Double
End Class
```

- **C#**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace FlexPieQuickStart
{
    class DataCreator
    {
        public static List<FruitDataItem> CreateFruit()
        {
            var fruits = new string[] { "Oranges", "Apples", "Pears", "Bananas" };
            var count = fruits.Length;
            var result = new List<FruitDataItem>();
            var rnd = new Random();
            for (var i = 0; i < count; i++)
                result.Add(new FruitDataItem()
                {
                    Fruit = fruits[i],
                    March = rnd.Next(20),
                    April = rnd.Next(20),
                    May = rnd.Next(20),
                });
            return result;
        }
    }

    public class FruitDataItem
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
    {
        public string Fruit { get; set; }
        public double March { get; set; }
        public double April { get; set; }
        public double May { get; set; }
    }
}
```

2. Edit the XAML markup to provide data to FlexPie.

   - **XAML**

```
<Window
         x:Class="FlexPieQuickStart.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:FlexPieQuickStart"
        mc:Ignorable="d"
        DataContext="{Binding RelativeSource={RelativeSource Mode=Self}}"
        xmlns:Chart="http://schemas.componentone.com/winfx/2006/xaml"
        Title="MainWindow" Height="800" Width="800">
    <Grid>

        <Chart:C1FlexPie
            x:Name="flexPie"
            Binding="April"
            BindingName="Fruit"
            ItemsSource="{Binding DataContext.Data}">
            <Chart:C1FlexPie.SelectionStyle>
                <Chart:ChartStyle Stroke="Red"
                                  StrokeThickness="2"/>
            </Chart:C1FlexPie.SelectionStyle>
            <Chart:C1FlexPie.DataLabel>
                <Chart:PieDataLabel Content="{}{y}"/>
            </Chart:C1FlexPie.DataLabel>
        </Chart:C1FlexPie>
    </Grid>
```

> To specify the binding source, you need to add the **DataContext = "{Binding RelativeSource= {RelativeSource Mode=Self}}"** markup in the **<Window>** tag of the **MainWindow.xaml** file.

3. Switch to **Code view** and add the following code.

   - **Visual Basic**

```
Partial Public Class MainWindow
    Inherits Window
    Private _data As List(Of FruitDataItem)
    Public Sub New()
        InitializeComponent()
    End Sub
    Public ReadOnly Property Data() As List(Of FruitDataItem)
        Get
            If _data Is Nothing Then
                _data = DataCreator.CreateFruit()
            End If
            Return _data
        End Get
    End Property
End Class
```

   - **C#**

```
using System;
using System.Collections.Generic;
using System.Linq;
```

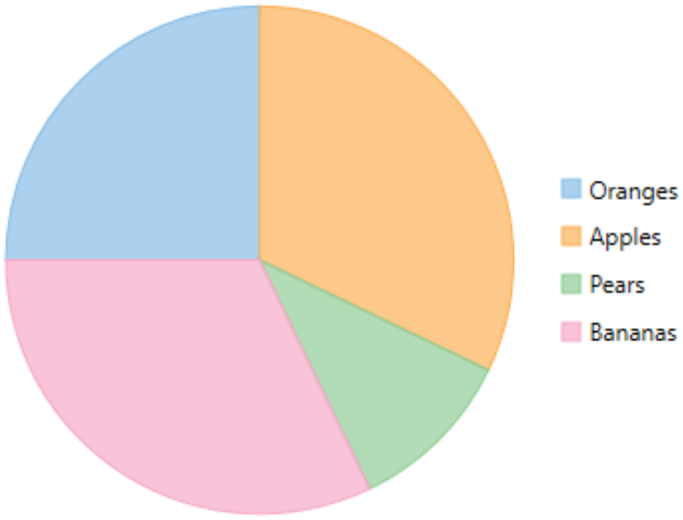
Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace FlexPieQuickStart
{
    public partial class MainWindow : Window
    {
        List<FruitDataItem> _data;
        public MainWindow()
        {
            InitializeComponent();
        }
        public List<FruitDataItem> Data
        {
            get
            {
                if (_data == null)
                {
                    _data = DataCreator.CreateFruit();
                }
                return _data;
            }
        }
    }
}
```

Step 3: Running the Application

Press F5 to run the application and observe the following output.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



Doughnut Pie Chart

FlexPie allows you to create the doughnut Pie Chart by using the InnerRadius property.

The inner radius is measured as a fraction of the radius of the Pie Chart. The default value of the InnerRadius property is zero, which creates the Pie Chart. Setting this property to values greater than zero creates the Pie Chart with a hole in the middle, also known as the Doughnut Chart.

The following code snippets demonstrate how to set the InnerRadius property.

XAML

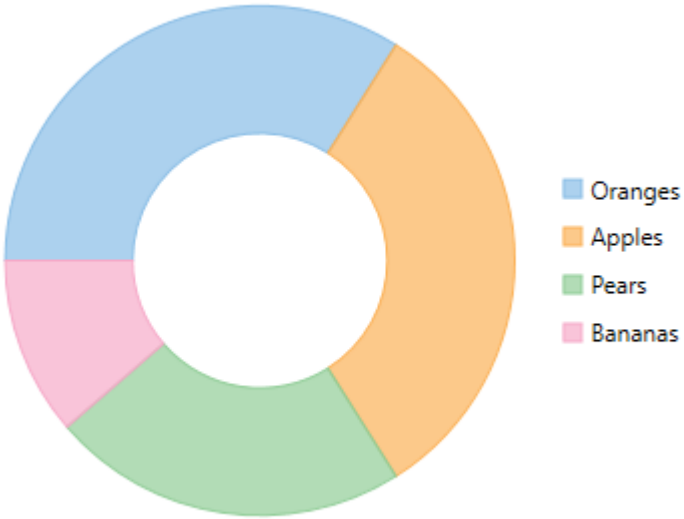
```
<Grid>

    <Chart:C1FlexPie
        x:Name="flexPie"
        InnerRadius="0.5"
        Offset="0.5"
        Header="Fruit Sales Data"
        Footer="2015"
        LegendOrientation="Horizontal"
        LegendPosition="Bottom"
        SelectionMode="Point"
        SelectedItemPosition="Top"
        SelectedItemOffset="2"
        Margin="95,99,328,275"
        Binding="April"
        BindingName="Fruit"
        ItemsSource="{Binding DataContext.Data}">
        <Chart:C1FlexPie.SelectionStyle>
            <Chart:ChartStyle Stroke="Red"
                              StrokeThickness="2"/>
        </Chart:C1FlexPie.SelectionStyle>
        <Chart:C1FlexPie.DataLabel>
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
            <Chart:PieDataLabel Content="{}{y}"/>
        </Chart:C1FlexPie.DataLabel>
    </Chart:C1FlexPie>
</Grid>
```

Code

C#	copyCode
`flexPie.InnerRadius = 0.5;`	



Exploded Pie Chart

The Offset property can be used to push the pie slices away from the center of **FlexPie**, producing the exploded Pie Chart. This property accepts a double value to determine how far the pie slices should be pushed from the center.

Here is the code snippet:

XAML

```
<Grid>

    <Chart:C1FlexPie
        x:Name="flexPie"
        InnerRadius="0.5"
        Offset="0.5"
        Header="Fruit Sales Data"
        Footer="2015"
        LegendOrientation="Horizontal"
        LegendPosition="Bottom"
        SelectionMode="Point"
        SelectedItemPosition="Top"
        SelectedItemOffset="2"
        Margin="95,99,328,275"
        Binding="April"
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
        BindingName="Fruit"
        ItemsSource="{Binding DataContext.Data}">
        <Chart:C1FlexPie.SelectionStyle>
            <Chart:ChartStyle Stroke="Red"
                              StrokeThickness="2"/>
        </Chart:C1FlexPie.SelectionStyle>
        <Chart:C1FlexPie.DataLabel>
            <Chart:PieDataLabel Content="{}{y}"/>
        </Chart:C1FlexPie.DataLabel>
    </Chart:C1FlexPie>
</Grid>
```

Code

C#	copyCode

```
flexPie.Offset = 0.5;
```

Animation

FlexPie allows you to control how the animation is applied to each series and series elements. It allows you to enable chart animation effects through a combination of different properties available in the FlexPie class. These properties allow you to apply duration, delay and an easing function for each animation. The animation effects are applied in one of the two scenarios, either while loading the chart for the first time or while the chart is redrawn after modifications.

Load

Implement animation when the data is first loaded into the chart, example when a new series is added to the FlexPie, you can apply animation properties. The following image shows how animation works while loading the FlexPie control.

The following image shows how animation works in the FlexPie control.

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Use the following code to implement animation in the FlexPie control.

C#

```
// Animation
flexPie1.AnimationSettings = C1.Chart.AnimationSettings.Load;
flexPie1.AnimationUpdate.Easing = C1.Chart.Easing.Linear;
flexPie1.AnimationUpdate.Duration = 500;
flexPie1.AnimationLoad.Type = C1.Chart.AnimationType.Series;
```

Every animation scenario has its own set of options that can be set independently in the FlexChart control. These options include various properties describing the corresponding animation.

To implement animation in the FlexChart control, you need make use of the following properties.

1. AnimationSettings -This property allows the user to apply settings for the animation. It allows the user to specify that when to apply animation in the FlexChart control. This property accepts values from the AnimationSettings enumeration provided by the FlexChart class. The AnimationSettings enumeration has special flags to control axes animation (smooth transition) so that you can enable or disable smooth axis transition for loading or updating data.
2. AnimationOptions - The AnimationLoad and AnimationUpdate properties includes the following options.
    1. Duration -This property allows you to set the duration of animation in the FlexChart control. This property accepts an integer value which defines duration in milliseconds.
    2. Easing - This property allows the user to set different type of easing functions on the FlexChart control. This property accepts values from the Easing enumeration provided by the C1.Chart namespace.
    3. Type – This property allows you to set the animation type on the FlexChart control. This property accepts the following values from the AnimationType enumeration provided by the C1.Chart

Copyright © 2019 GrapeCity, Inc. All rights reserved.

namespace.

- **All** - All plot elements animate at once from the bottom of the plot area.
- **Series** - Each series animates one at a time from the bottom of the plot area.
- **Points** - The plot elements appear one at a time from left to right.

Header & Footer

You can add a header to the **FlexPie** control by setting the Header property of FlexChartBase. Besides a header, you can add a footer to the control by setting the Footer property of FlexChartBase.

See the following code snippet for setting the two properties:

XAML

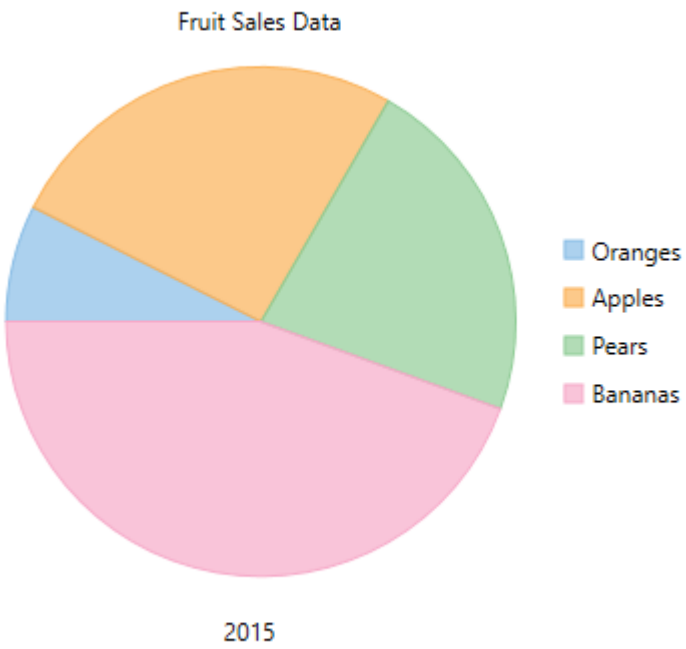
```
<Grid>

    <Chart:C1FlexPie
        x:Name="flexPie"
        InnerRadius="0.5"
        Offset="0.5"
        Header="Fruit Sales Data"
        Footer="2015"
        LegendOrientation="Horizontal"
        LegendPosition="Bottom"
        SelectionMode="Point"
        SelectedItemPosition="Top"
        SelectedItemOffset="2"
        Margin="95,99,328,275"
        Binding="April"
        BindingName="Fruit"
        ItemsSource="{Binding DataContext.Data}">
        <Chart:C1FlexPie.SelectionStyle>
            <Chart:ChartStyle Stroke="Red"
                              StrokeThickness="2"/>
        </Chart:C1FlexPie.SelectionStyle>
        <Chart:C1FlexPie.DataLabel>
            <Chart:PieDataLabel Content="{}{y}"/>
        </Chart:C1FlexPie.DataLabel>
    </Chart:C1FlexPie>
</Grid>
```

Code

C# copyCode

```
flexPie.Header = "Fruit Sales Data";
flexPie.Footer = "2015";
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.



Legend

FlexPie enables you to specify the position where you want to display the Legend by using the Legend property of FlexChartBase.

The below-mentioned code snippet shows how to set the property.

XAML

```
<Grid>

    <Chart:C1FlexPie
        x:Name="flexPie"
        InnerRadius="0.5"
        Offset="0.5"
        Header="Fruit Sales Data"
        Footer="2015"
        LegendOrientation="Horizontal"
        LegendPosition="Bottom"
        SelectionMode="Point"
        SelectedItemPosition="Top"
        SelectedItemOffset="2"
        Margin="95,99,328,275"
        Binding="April"
        BindingName="Fruit"
        ItemsSource="{Binding DataContext.Data}">
        <Chart:C1FlexPie.SelectionStyle>
            <Chart:ChartStyle Stroke="Red"
                              StrokeThickness="2"/>
        </Chart:C1FlexPie.SelectionStyle>
        <Chart:C1FlexPie.DataLabel>
```

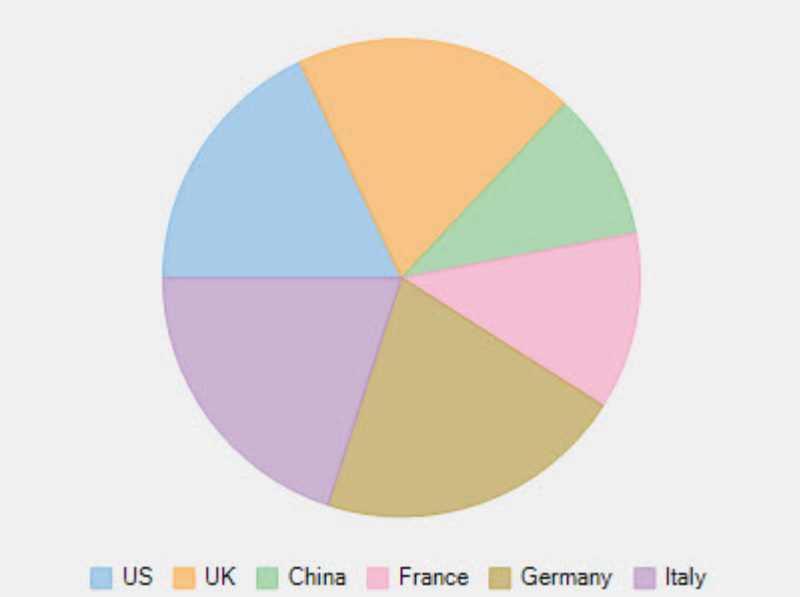
Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
            <Chart:PieDataLabel Content="{}{y}"/>
        </Chart:C1FlexPie.DataLabel>
    </Chart:C1FlexPie>
</Grid>
```

Code

C# copyCode

```
flexPie.LegendOrientation = C1.Chart.Orientation.Horizontal;
flexPie.LegendPosition = C1.Chart.Position.Bottom;
```



Selection

You can choose what element of **FlexPie** is selected when you click anywhere on the control by setting the SelectionMode property. This property provides three options:

- **None**: Does not select any element.
- **Point**: Highlights the pie slice that the user clicks.
- **Series**: Highlights the entire pie.

After setting the SelectionMode property to Point, you can change the position of the selected pie slice by setting the SelectedItemPosition property. And also, you can move the selected pie slice away from the center of FlexPie by setting the SelectedItemOffset property.

XAML

```
<Grid>

    <Chart:C1FlexPie
        x:Name="flexPie"
        InnerRadius="0.5"
        Offset="0.5"
        Header="Fruit Sales Data"
```

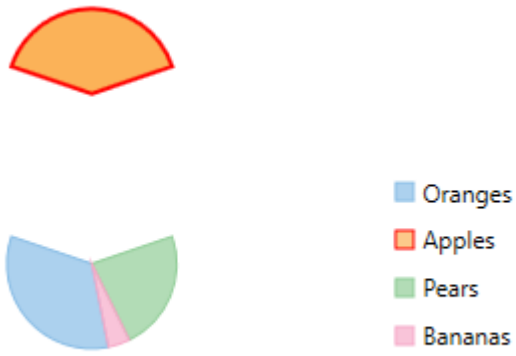
Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
        Footer="2015"
        LegendOrientation="Horizontal"
        LegendPosition="Bottom"
        SelectionMode="Point"
        SelectedItemPosition="Top"
        SelectedItemOffset="2"
        Margin="95,99,328,275"
        Binding="April"
        BindingName="Fruit"
        ItemsSource="{Binding DataContext.Data}">
        <Chart:C1FlexPie.SelectionStyle>
            <Chart:ChartStyle Stroke="Red"
                              StrokeThickness="2"/>
        </Chart:C1FlexPie.SelectionStyle>
        <Chart:C1FlexPie.DataLabel>
            <Chart:PieDataLabel Content="{}{y}"/>
        </Chart:C1FlexPie.DataLabel>
    </Chart:C1FlexPie>
</Grid>
```

Code

C# copyCode

```
flexPie.SelectionMode = C1.Chart.ChartSelectionMode.Point;
flexPie.SelectedItemPosition = C1.Chart.Position.Top;
flexPie.SelectedItemOffset = 2;
```



Data Labels

Data labels provide additional information about the data points. These labels make a chart easier to understand because they show details about a slice in the pie.

To understand the working of data labels in FlexPie chart, refer to the following sections.

Adding and Positioning Data Labels
: Learn how to add data labels and set their position on the chart.

Formatting Data Labels
: Learn how to perform styling and formatting of data labels.

Managing Overlapped Data Labels
: Learn how to manage overlapping data labels in FlexPie chart.

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Adding and Positioning Labels

With FlexPie chart, you can configure the arrangement and display properties for data labels depending on what suits your needs the best. By default, the data labels are not displayed on the chart, however, you can enable them by setting the Position and Content properties of DataLabel class. Here, the **Position** property sets position of the data labels by accepting values from the PieLabelPosition enumeration and the **Content** property gets or sets content for the data labels.

The example code below uses the Position and Content properties to enable data labels and set their position.

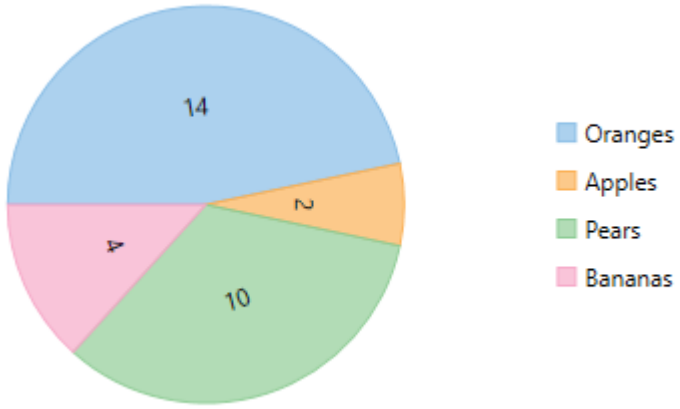
XAML

```
<c1:C1FlexPie.DataLabel> <c1:PieDataLabel Content="{}{y}" Position="Circular"/>
</c1:C1FlexPie.DataLabel>
```

Code

HTML

```
flexPie.DataLabel.Content = "{}{y}";
flexPie.DataLabel.Position = C1.Chart.PieLabelPosition.Circular;
```



Formatting Data Labels

FlexPie provides various options to format data labels according to your requirements. You can use connecting lines to connect the data labels, set and style borders of data labels, and customize the appearance of data labels.

The topic comprises of three sections:

- **Setting and Styling Borders**
- **Connecting DataLabels to Data Points**
- **Modifying Appearance**

Setting and Styling Borders

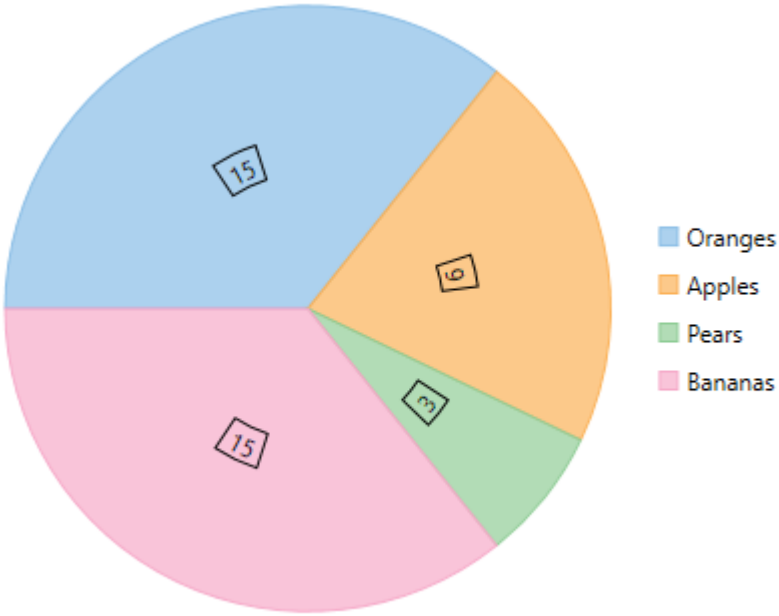
To add and style borders to FlexPie data labels, set the Border and BorderStyle properties provided by DataLabel class.

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Use the following code snippet to add borders to data labels of FlexPie.

C#

```
// Enable Border
flexPie.DataLabel.Border = true;
```



Connecting DataLabels to Data Points

In case the data labels are placed away from the data points, you can connect them using connecting lines.

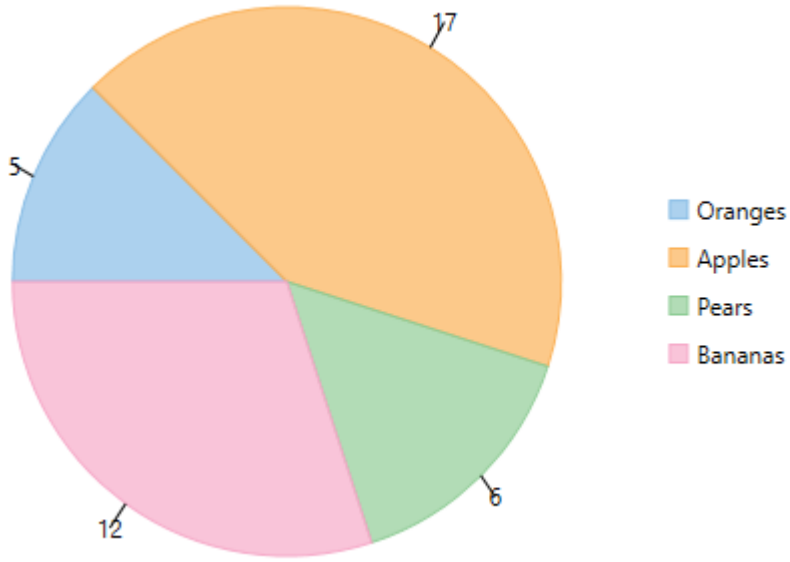
To enable connecting lines in FlexPie chart, you need to use the ConnectingLine property.

Use the following code snippet to set the connecting lines.

C#

```
//Enable connecting line
flexPie.DataLabel.ConnectingLine = true;
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.



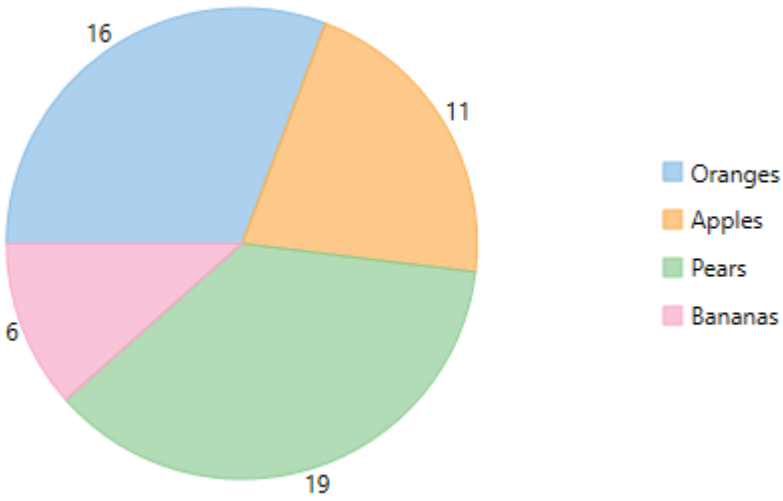
Modifying Appearance

FlexPie includes various styling options, to enhance the clarity of data labels. To modify the appearance of FlexPie chart, you need to use the Style property. It allows you to modify the font family, fill color, use stroke brush for data labels, set width for stroke brush and more.

In the example code, we have modified the font used in the chart and set the stroke width property. Use the following code snippet to modify the appearance of the chart.

C#

```
// Modifying appearance of data labels
flexPie.DataLabel.Style.FontFamily = new FontFamily("GenericSerif");
flexPie.DataLabel.Style.StrokeThickness = 2;
```



Manage Overlapped Data Labels

Copyright © 2019 GrapeCity, Inc. All rights reserved.

A common issue pertaining to charts is overlapping of data labels that represent data points. In most cases, overlapping occurs due to long text in data labels or large numbers of data points.

To manage overlapped data labels in FlexPie chart, you can make use of Overlapping property provided by PieDataLabel class. The Overlapping property accepts the following values from the PieLabelOverlapping enumeration.

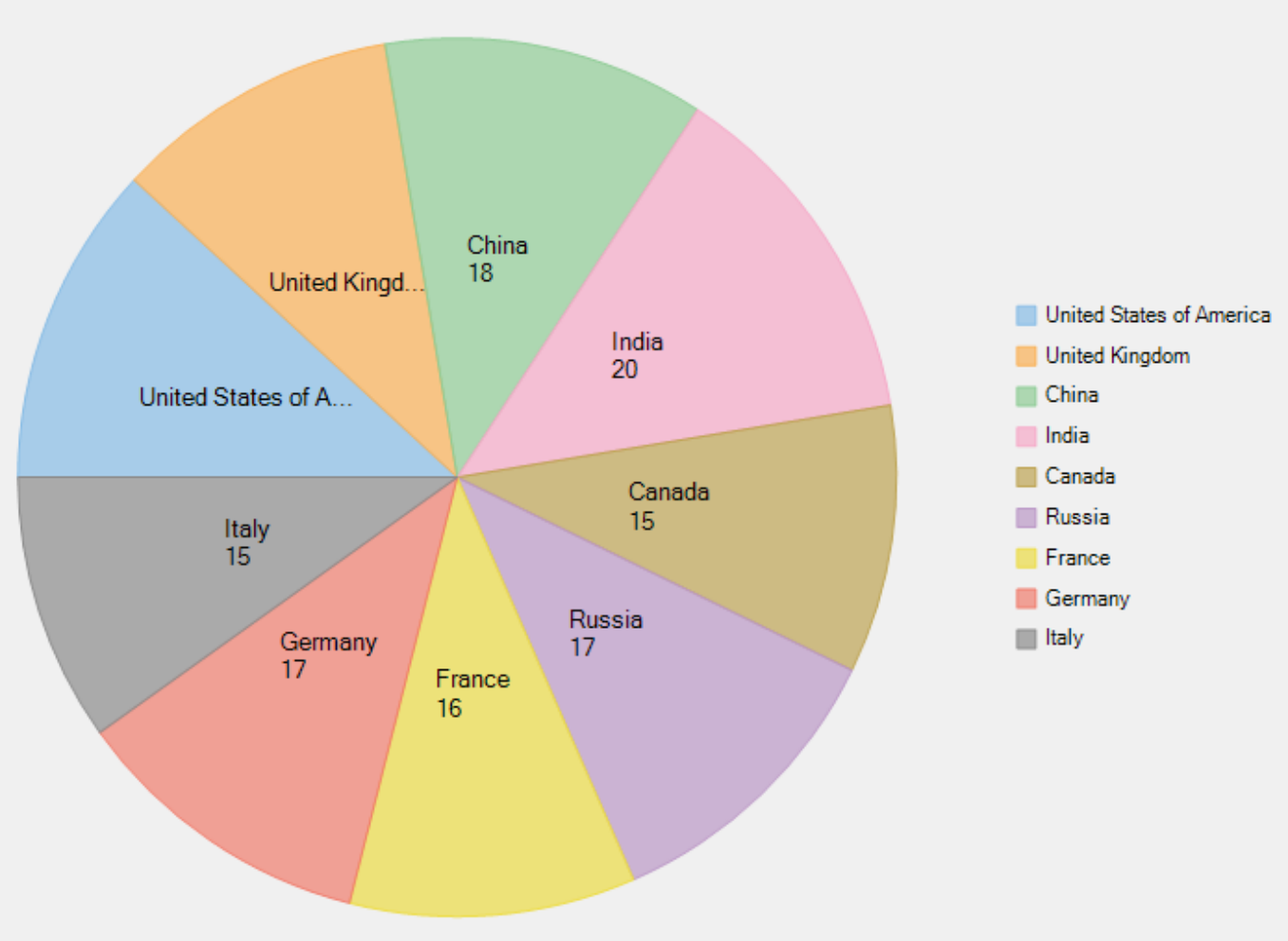
Enumeration	Description
PieLabelOverlapping.Default	Show all labels including the overlapping ones.
PieLabelOverlapping.Hide	Hides the overlapping labels, if its content is larger than the corresponding pie segment.
PieLabelOverlapping.Trim	Trim overlapping data labels, if its width is larger than the corresponding pie segment.

Use the following code to manage overlapping data labels.

C#

```
//Set Overlapping property
flexPie.DataLabel.Overlapping = C1.Chart.PieLabelOverlapping.Trim;
```

The following image shows how FlexPie appears after setting the Overlapping property.



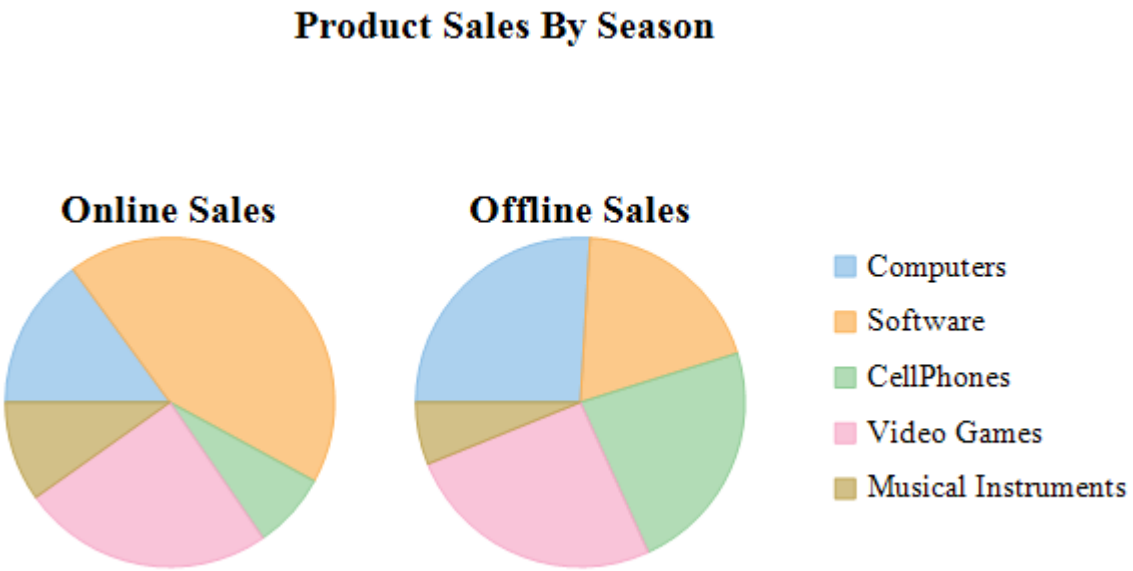
Copyright © 2019 GrapeCity, Inc. All rights reserved.

Multiple Pie Charts

In some scenarios, you might want to visualize a bit more than a single pie chart, and use multiple Pie Charts together. The **FlexPie** control empowers a user to create multiple Pie Charts based on the same data source. The user can create multiple pie charts by simply specifying several comma-separated strings of field names in the Binding property of C1FlexPie class. This means you don't have to add any additional FlexPie controls for multiple Pie Charts.

Multiple Pie Charts can help compare data across different groups. It is also the right visualization choice to present key indicators/facts about your topic of interest.

The following image shows two Pie Charts, Online Sales and Offline Sales for five different products. Computer, Software, CellPhones, Video Games and Musical Instruments.



Adding multiple pie charts using the same data source is depicted programmatically below:

XAML

Code

```
<Window.Resources>
        <x:Array Type="{x:Type sys:String}" x:Key="MyMultipleTitles">
            <sys:String>Online Sales</sys:String>
            <sys:String>Offline Sales</sys:String>
        </x:Array>
    </Window.Resources>
    <Grid>
        <c1:C1FlexPie x:Name="flexPie" ItemsSource="{Binding}"
Header="Products Sales By Season" Binding="Online,Offline" BindingName="Name"
HorizontalAlignment="Left" Height="300" Margin="233,69,0,0" VerticalAlignment="Top"
Width="300" Titles="{StaticResource MyMultipleTitles}">
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
        </c1:C1FlexPie>
    </Grid>
</Window>
```

C#

Code

```
flexPie.ItemsSource = data;
            flexPie.BindingName = "Name";
            flexPie.Binding = "Online,Offline";
            flexPie.Titles = new[] { "Online Sales", "Offline Sales" };
            flexPie.Header = "Product Sales By Season";
```

Visual Basic

Code

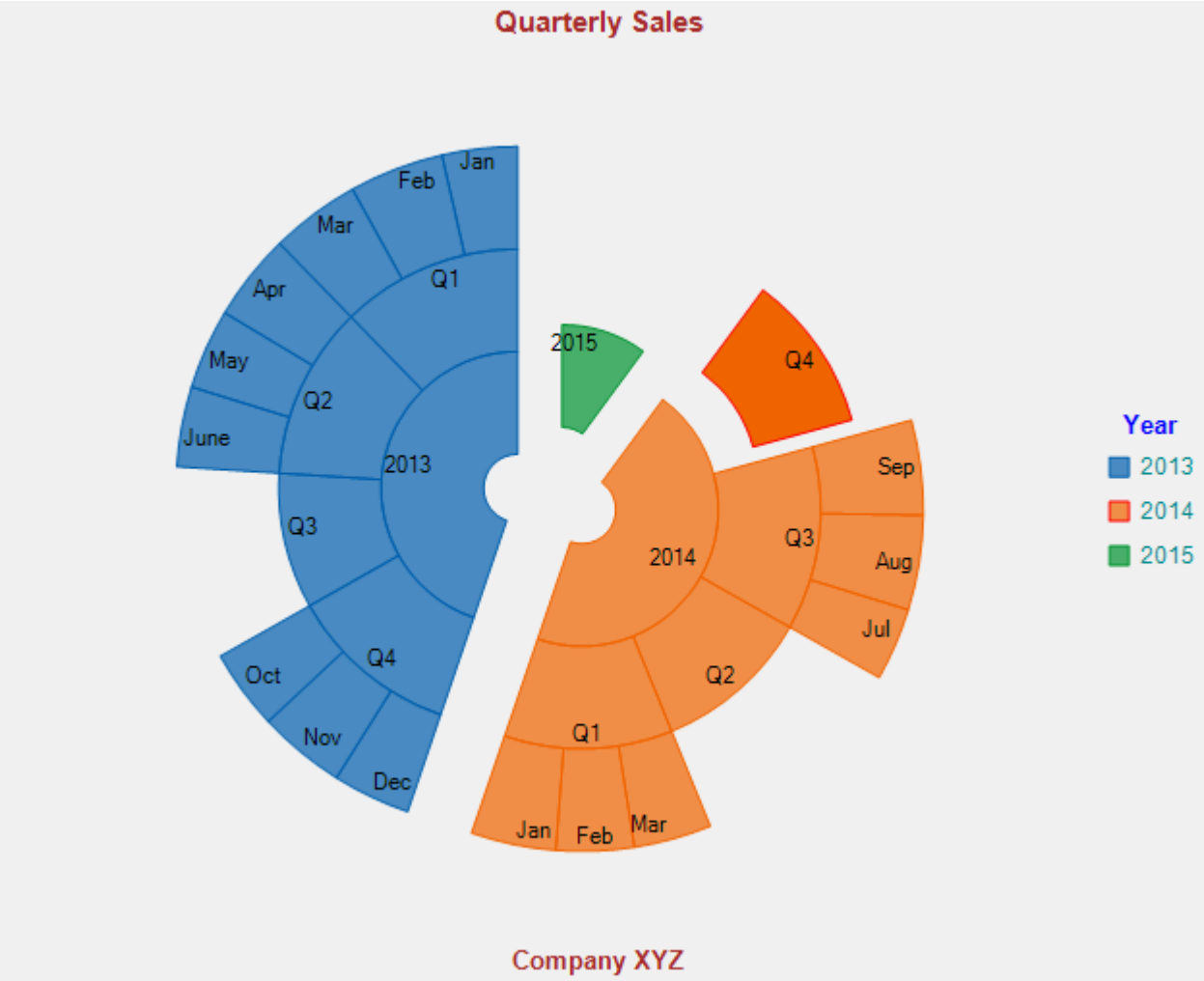
```
 Private Sub Window_Loaded(sender As Object, e As RoutedEventArgs)
         flexPie.ItemsSource = _data
         flexPie.BindingName = "Name"
         flexPie.Binding = "Online,Offline"
        flexPie.Titles = {"Online Sales", "Offline Sales"}
         flexPie.Header = "Product Sales By Season"
     End Sub
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Sunburst Chart

Sunburst, also known as a multi-level pie chart, is ideal for visualizing multi-level hierarchical data depicted by concentric circles.The circle in the center represents the root node, with the data moving outside from the center. A section of the inner circle supports a hierarchical relationship to those sections of the outer circle which lie within the angular area of the parent section.

Using Sunburst chart helps the end user to visualize the relationship between outer rings and inner rings. For instance, you might want to display sales report for each quarter of three years. Using SunBurst chart, a sales report of a specific month can be highlighted, thereby depicting relationship with the respective quarter.



To explore the features of Sunburst chart, click the following links:

- Quick Start
- Key Features
- Legend and Titles
- Selection
- Drilldown

Quick Start

This quick start is intended to guide you through a step-by-step process of creating a simple Sunburst application and running the same in Visual Studio.

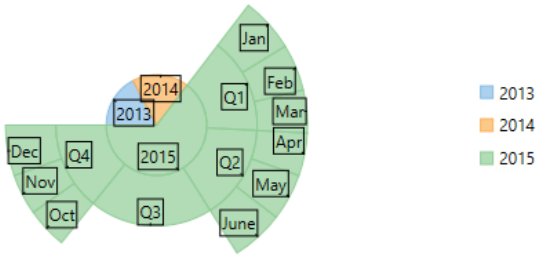
To quickly get started with Sunburst chart and observe how it appears on running the application, follow these steps:

1. **Add Sunburst Chart to the Application**
2. **Bind Sunburst Chart to a Data Source**

Copyright © 2019 GrapeCity, Inc. All rights reserved.

3. **Run the Application**

The following image displays how a basic Sunburst chart appears after completing the steps mentioned above.



Step 1: Add Sunburst Chart to the Application

1. Create a **WPF Application** in Visual Studio.
2. Drag and drop the **C1Sunburst** control to the MainWindow.
   The following dlls are automatically added to the application:
   C1.WPF.4.dll
   C1.WPF.DX.4.dll
   C1.WPF.FlexChart.4.dll

   The XAML markup resembles the following code in the **<Grid></Grid>** tags.
   - **XAML**

```
<c1:C1Sunburst x:Name="flexPie"
               Binding="Value"
               BindingName="Name"
               HorizontalAlignment="Left"
               Height="300"
               VerticalAlignment="Top"
               Width="300">
    <c1:C1Sunburst.ItemsSource>
        <c1:FlexPieSliceCollection>
            <c1:FlexPieSlice Name="Slice1" Value="1"/>
            <c1:FlexPieSlice Name="Slice2" Value="2"/>
            <c1:FlexPieSlice Name="Slice3" Value="3"/>
            <c1:FlexPieSlice Name="Slice4" Value="4"/>
        </c1:FlexPieSliceCollection>
    </c1:C1Sunburst.ItemsSource>
</c1:C1Sunburst>
```

Step 2: Bind Sunburst Chart to a Data Source

In this step, first create a class DataService that generates random sales data for four quarters, namely Q1, Q2, Q3, and Q4 in 2013, 2014, and 2015. Next, bind Sunburst to the created class using the ItemsSource property provided by the FlexChartBase class. Then, specify numeric values and labels for the Sunburst slices using the Binding and the BindingName property, respectively of the FlexChartBase and the C1FlexPie class.

1. Add a class, **DataService** and add the following code.
   - **Visual Basic**

```
Imports System.Collections.Generic
Imports System.Linq
Imports System.Text
Imports System.Threading.Tasks

Public Class DataService
    Private rnd As New Random()
    Shared _default As DataService

    Public Shared ReadOnly Property Instance() As DataService
        Get
            If _default Is Nothing Then
                _default = New DataService()
            End If

            Return _default
        End Get
    End Property

    Public Shared Function CreateHierarchicalData() As List(Of DataItem)
        Dim rnd As Random = Instance.rnd

        Dim years As New List(Of String)()
        Dim times As New List(Of List(Of String))() From {
            New List(Of String)() From {
                "Jan",
                "Feb",
                "Mar"
            },
            New List(Of String)() From {
                "Apr",
                "May",
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
            "June"
        },
        New List(Of String)() From {
            "Jul",
            "Aug",
            "Sep"
        },
        New List(Of String)() From {
            "Oct",
            "Nov",
            "Dec"
        }
    }

    Dim items As New List(Of DataItem)()
    Dim yearLen = Math.Max(CInt(Math.Round(Math.Abs(5 - Instance.rnd.NextDouble() * 10))), 3)
    Dim currentYear As Integer = DateTime.Now.Year
    For i As Integer = yearLen To 1 Step -1
        years.Add((currentYear - i).ToString())
    Next
    Dim quarterAdded = False

    years.ForEach(
        Function(y)
            Dim i = years.IndexOf(y)
            Dim addQuarter = Instance.rnd.NextDouble() > 0.5
            If Not quarterAdded AndAlso i = years.Count - 1 Then
                addQuarter = True
            End If
            Dim year = New DataItem() With {
                        .Year = y
                    }
            If addQuarter Then
                quarterAdded = True
                times.ForEach(Function(q)
                                  Dim addMonth = Instance.rnd.NextDouble() > 0.5
                                  Dim idx As Integer = times.IndexOf(q)
                                  Dim quar As String = "Q" + (idx + 1).ToString()
                                  Dim quarters = New DataItem() With {
                                              .Year = y,
                                              .Quarter = quar
                                          }
                                  If addMonth Then
                                      q.ForEach(
                                      Function(m)
                                          quarters.Items.Add(New DataItem() With {
                                                                        .Year = y,
                                                                        .Quarter = quar,
                                                                        .Month = m,
                                                                        .Value = rnd.[Next](20, 30)
                                                                    })

                                      End Function)
                                  Else
                                      quarters.Value = rnd.[Next](80, 100)
                                  End If
                                  year.Items.Add(quarters)

                              End Function)
            Else
                year.Value = rnd.[Next](80, 100)
            End If
            items.Add(year)

        End Function)

    Return items
End Function

Public Shared Function CreateFlatData() As List(Of FlatDataItem)
    Dim rnd As Random = Instance.rnd
    Dim years As New List(Of String)()
    Dim times As New List(Of List(Of String))() From {
        New List(Of String)() From {
            "Jan",
            "Feb",
            "Mar"
        },
        New List(Of String)() From {
            "Apr",
            "May",
            "June"
        },
        New List(Of String)() From {
            "Jul",
            "Aug",
            "Sep"
        },
        New List(Of String)() From {
            "Oct",
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
                "Nov",
                "Dec"
            }
        }

        Dim items As New List(Of FlatDataItem)()
        Dim yearLen = Math.Max(CInt(Math.Round(Math.Abs(5 - rnd.NextDouble() * 10))), 3)
        Dim currentYear As Integer = DateTime.Now.Year
        For i As Integer = yearLen To 1 Step -1
            years.Add((currentYear - i).ToString())
        Next
        Dim quarterAdded = False
        years.ForEach(
            Function(y)
                Dim i = years.IndexOf(y)
                Dim addQuarter = rnd.NextDouble() > 0.5
                If Not quarterAdded AndAlso i = years.Count - 1 Then
                    addQuarter = True
                End If
                If addQuarter Then
                    quarterAdded = True
                    times.ForEach(Function(q)
                                      Dim addMonth = rnd.NextDouble() > 0.5
                                      Dim idx As Integer = times.IndexOf(q)
                                      Dim quar As String = "Q" + (idx + 1).ToString()
                                      If addMonth Then
                                          q.ForEach(Function(m)
                                                        items.Add(New FlatDataItem() With {
                                                                  .Year = y,
                                                                  .Quarter = quar,
                                                                  .Month = m,
                                                                  .Value = rnd.[Next](30, 40)
                                                              })

                                                    End Function)
                                      Else
                                          items.Add(New FlatDataItem() With {
                                                        .Year = y,
                                                        .Quarter = quar,
                                                        .Value = rnd.[Next](80, 100)
                                                    })
                                      End If

                                  End Function)
                Else
                    items.Add(New FlatDataItem() With {
                                  .Year = y.ToString(),
                                  .Value = rnd.[Next](80, 100)
                              })
                End If

            End Function)

        Return items
    End Function
End Class
```

- **C#**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;


namespace SunburstQuickStart
{
    public class DataService
    {
        Random rnd = new Random();
        static DataService _default;

        public static DataService Instance
        {
            get
            {
                if (_default == null)
                {
                    _default = new DataService();
                }

                return _default;
            }
        }

        public static List<DataItem> CreateHierarchicalData()
        {
            Random rnd = Instance.rnd;

            List<string> years = new List<string>();
            List<List<string>> times = new List<List<string>>()
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
    {
        new List<string>() { "Jan", "Feb", "Mar"},
        new List<string>() { "Apr", "May", "June"},
        new List<string>() { "Jul", "Aug", "Sep"},
        new List<string>() { "Oct", "Nov", "Dec" }
    };

    List<DataItem> items = new List<DataItem>();
    var yearLen = Math.Max((int)Math.Round(Math.Abs(5 - Instance.rnd.NextDouble() * 10)), 3);
    int currentYear = DateTime.Now.Year;
    for (int i = yearLen; i > 0; i--)
    {
        years.Add((currentYear - i).ToString());
    }
    var quarterAdded = false;

    years.ForEach(y =>
    {
        var i = years.IndexOf(y);
        var addQuarter = Instance.rnd.NextDouble() > 0.5;
        if (!quarterAdded && i == years.Count - 1)
        {
            addQuarter = true;
        }
        var year = new DataItem() { Year = y };
        if (addQuarter)
        {
            quarterAdded = true;
            times.ForEach(q =>
            {
                var addMonth = Instance.rnd.NextDouble() > 0.5;
                int idx = times.IndexOf(q);
                var quar = "Q" + (idx + 1);
                var quarters = new DataItem() { Year = y, Quarter = quar };
                if (addMonth)
                {
                    q.ForEach(m =>
                    {
                        quarters.Items.Add(new DataItem()
                        {
                            Year = y,
                            Quarter = quar,
                            Month = m,
                            Value = rnd.Next(20, 30)
                        });
                    });
                }
                else
                {
                    quarters.Value = rnd.Next(80, 100);
                }
                year.Items.Add(quarters);
            });
        }
        else
        {
            year.Value = rnd.Next(80, 100);
        }
        items.Add(year);
    });

    return items;
}

public static List<FlatDataItem> CreateFlatData()
{
    Random rnd = Instance.rnd;
    List<string> years = new List<string>();
    List<List<string>> times = new List<List<string>>()
    {
        new List<string>() { "Jan", "Feb", "Mar"},
        new List<string>() { "Apr", "May", "June"},
        new List<string>() { "Jul", "Aug", "Sep"},
        new List<string>() { "Oct", "Nov", "Dec" }
    };

    List<FlatDataItem> items = new List<FlatDataItem>();
    var yearLen = Math.Max((int)Math.Round(Math.Abs(5 - rnd.NextDouble() * 10)), 3);
    int currentYear = DateTime.Now.Year;
    for (int i = yearLen; i > 0; i--)
    {
        years.Add((currentYear - i).ToString());
    }
    var quarterAdded = false;
    years.ForEach(y =>
    {
        var i = years.IndexOf(y);
        var addQuarter = rnd.NextDouble() > 0.5;
        if (!quarterAdded && i == years.Count - 1)
        {
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
                    addQuarter = true;
                }
                if (addQuarter)
                {
                    quarterAdded = true;
                    times.ForEach(q =>
                    {
                        var addMonth = rnd.NextDouble() > 0.5;
                        int idx = times.IndexOf(q);
                        var quar = "Q" + (idx + 1);
                        if (addMonth)
                        {
                            q.ForEach(m =>
                            {
                                items.Add(new FlatDataItem()
                                {
                                    Year = y,
                                    Quarter = quar,
                                    Month = m,
                                    Value = rnd.Next(30, 40)
                                });
                            });
                        }
                        else
                        {
                            items.Add(new FlatDataItem()
                            {
                                Year = y,
                                Quarter = quar,
                                Value = rnd.Next(80, 100)
                            });
                        }
                    });
                }
                else
                {
                    items.Add(new FlatDataItem()
                    {
                        Year = y.ToString(),
                        Value = rnd.Next(80, 100)
                    });
                }
            });

            return items;
        }
    }
}
```

2. Add a class, **SunburstViewModel** and add the following code.
    - **Visual Basic**

```
Imports C1.Chart
Imports System.Linq
Imports System.Collections.Generic
Public Class SunburstViewModel

    Public ReadOnly Property HierarchicalData() As List(Of DataItem)
        Get
            Return DataService.CreateHierarchicalData()
        End Get
    End Property

    Public ReadOnly Property FlatData() As List(Of FlatDataItem)
        Get
            Return DataService.CreateFlatData()
        End Get
    End Property

    Public ReadOnly Property Positions() As List(Of String)
        Get
            Return [Enum].GetNames(GetType(Position)).ToList()
        End Get
    End Property

    Public ReadOnly Property Palettes() As List(Of String)
        Get
            Return [Enum].GetNames(GetType(Palette)).ToList()
        End Get
    End Property
End Class
```

    - **C#**

```
using C1.Chart;
using System;
using System.Linq;
using System.Collections.Generic;

namespace SunburstQuickStart
{
    public class SunburstViewModel
    {
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
        public List<DataItem> HierarchicalData
        {
            get
            {
                return DataService.CreateHierarchicalData();
            }
        }

        public List<FlatDataItem> FlatData
        {
            get
            {
                return DataService.CreateFlatData();
            }
        }

        public List<string> Positions
        {
            get
            {
                return Enum.GetNames(typeof(Position)).ToList();
            }
        }

        public List<string> Palettes
        {
            get
            {
                return Enum.GetNames(typeof(Palette)).ToList();
            }
        }
    }
}
```

3. Add a class, **DataItem** and add the following code.
    - **Visual Basic**

```
Public Class DataItem
    Private _items As List(Of DataItem)

    Public Property Year() As String
        Get
            Return m_Year
        End Get
        Set
            m_Year = Value
        End Set
    End Property
    Private m_Year As String
    Public Property Quarter() As String
        Get
            Return m_Quarter
        End Get
        Set
            m_Quarter = Value
        End Set
    End Property
    Private m_Quarter As String
    Public Property Month() As String
        Get
            Return m_Month
        End Get
        Set
            m_Month = Value
        End Set
    End Property
    Private m_Month As String
    Public Property Value() As Double
        Get
            Return m_Value
        End Get
        Set
            m_Value = Value
        End Set
    End Property
    Private m_Value As Double
    Public ReadOnly Property Items() As List(Of DataItem)
        Get
            If _items Is Nothing Then
                _items = New List(Of DataItem)()
            End If

            Return _items
        End Get
    End Property
End Class

Public Class FlatDataItem
    Public Property Year() As String
        Get
            Return m_Year
        End Get
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
        Set
            m_Year = Value
        End Set
    End Property
    Private m_Year As String
    Public Property Quarter() As String
        Get
            Return m_Quarter
        End Get
        Set
            m_Quarter = Value
        End Set
    End Property
    Private m_Quarter As String
    Public Property Month() As String
        Get
            Return m_Month
        End Get
        Set
            m_Month = Value
        End Set
    End Property
    Private m_Month As String
    Public Property Value() As Double
        Get
            Return m_Value
        End Get
        Set
            m_Value = Value
        End Set
    End Property
    Private m_Value As Double
End Class
```

- **C#**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SunburstQuickStart
{
    public class DataItem
    {
        List<DataItem> _items;

        public string Year { get; set; }
        public string Quarter { get; set; }
        public string Month { get; set; }
        public double Value { get; set; }
        public List<DataItem> Items
        {
            get
            {
                if (_items == null)
                {
                    _items = new List<DataItem>();
                }

                return _items;
            }
        }
    }

    public class FlatDataItem
    {
        public string Year { get; set; }
        public string Quarter { get; set; }
        public string Month { get; set; }
        public double Value { get; set; }
    }
}
```

4. Add a class, **Converter** and add the following code.
    - **Visual Basic**

```
Imports C1.Chart
Imports System.Collections.Generic
Imports System.Globalization
Imports System.Linq
Imports System.Text
Imports System.Threading.Tasks
Imports System.Windows.Data
Public Class EnumToStringConverter
    Implements IValueConverter
    Public Function Convert(value As Object,
                            targetType As Type,
                            parameter As Object,
                            culture As CultureInfo) As Object
        Return value.ToString()
    End Function
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
    Public Function ConvertBack(value As Object,
                                targetType As Type,
                                parameter As Object,
                                culture As CultureInfo) As Object
        If targetType = GetType(Position) Then
            Return DirectCast([Enum].Parse(GetType(Position),
                                           value.ToString()),
                                           Position)
        Else
            Return DirectCast([Enum].Parse(GetType(Palette),
                                           value.ToString()),
                                           Palette)
        End If
    End Function

    Private Function IValueConverter_Convert(value As Object,
                                             targetType As Type,
                                             parameter As Object,
                                             culture As CultureInfo) As Object Implements IValueConverter.Convert
        Throw New NotImplementedException()
    End Function

    Private Function IValueConverter_ConvertBack(value As Object,
                                                 targetType As Type,
                                                 parameter As Object,
                                                 culture As CultureInfo) As Object Implements IValueConverter.ConvertBack
        Throw New NotImplementedException()
    End Function
End Class
```

- **C#**

```
using C1.Chart;
using System;
using System.Collections.Generic;
using System.Globalization;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Data;

namespace SunburstQuickStart
{
    public class EnumToStringConverter : IValueConverter
    {
        public object Convert(object value, Type targetType, object parameter, CultureInfo culture)
        {
            return value.ToString();
        }

        public object ConvertBack(object value, Type targetType, object parameter, CultureInfo culture)
        {
            if (targetType == typeof(Position))
                return (Position)Enum.Parse(typeof(Position), value.ToString());
            else
                return (Palette)Enum.Parse(typeof(Palette), value.ToString());
        }
    }
}
```

5. Edit the XAML code to provide data to Sunburst.

- **XAML**

```
<Window
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:SunburstQuickStart"
        xmlns:c1="http://schemas.componentone.com/winfx/2006/xaml"
        x:Class="SunburstQuickStart.MainWindow"
        mc:Ignorable="d"
        Title="MainWindow" Height="800" Width="691.337">
    <Grid Margin="0,0,0,-120">
        <Grid.Resources>
            <local:EnumToStringConverter x:Key="PaletteConverter" />
        </Grid.Resources>
        <Grid.DataContext>
            <local:SunburstViewModel />
        </Grid.DataContext>
        <c1:C1Sunburst x:Name="sunburst"
                       Offset="0"
                       ItemsSource="{Binding HierarchicalData}"
                       Binding="Value"
                       BindingName="Year,Quarter,Month"
                       ChildItemsPath="Items"
                       ToolTipContent="{}{name}&#x000A;{y}"
                       Margin="0,121,73,0"  >
            <c1:C1Sunburst.DataLabel>
                <c1:PieDataLabel Position="Inside"
                                 Content="{}{name}"
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
                                ConnectingLine="True"
                                Border="True">
                    </c1:PieDataLabel>
                </c1:C1Sunburst.DataLabel>
            </c1:C1Sunburst>
        </Grid>
    </Window>
```

Step 3: Running the Application

Press F5 to run the application and observe how Sunburst chart appears.

Key Features

The Sunburst control includes a number of features that make it efficient and useful for creating professional-looking applications.

- **Doughnut Sunburst chart**: Create a doughnut Sunburst chart by setting the InnerRadius property that has a default value of zero. Setting this property to a value greater than zero creates a hole in the middle, thereby creating the doughnut Sunburst chart.
- **Exploded Sunburst chart**: Create an exploded Sunburst chart by setting the Offset property that has a default value of zero. Setting the property pushes the slices away from the center of Sunburst chart, thereby producing the exploded Sunburst chart.
- **Reversed Sunburst chart**: Create a reversed Sunburst chart by setting the Reversed property that has False as the default value. Setting the property to True creates the reversed Sunburst chart that contains angles drawn in the counter-clockwise direction.
- **Start angle:** Set the start angle by setting the StartAngle property that accepts values of the double type. A start angle is the angle that is set in degrees to start drawing Sunburst slices in the clockwise direction. The default value is the 9 o'clock position.
- **Palette**: Use different color palettes to make Sunburst chart more appealing and presentable. To specify the chart palette, set the Palette property that lets you specify an array of default colors to be used when rendering slices. The property accepts values from the Palette enumeration.
- **Legend**: Perform various customizations including setting orientation, position, or styling the legend of Sunburst chart. For more information, refer to Legend and Titles.
- **Header and Footer**: Use simple properties to set and customize Header and Footer of Sunburst chart. For more information, refer to Legend and Titles.
- **Selection**: Change the selection mode and customize the position, appearance of the selected pie slice. For more information, refer to Selection.
- **Data Labels**: Add, style, format, set the position of data labels and manage the overlapped data labels on the chart. For more information, refer Data Labels.

Legend and Titles

Legend

The legend displays entries for series with their names and predefined symbols. In Sunburst, you can perform various customizations with the legend, as follows:

- **Orientation**: Set the orientation of the legend as horizontal, vertical, or automatic by using the LegendOrientation property provided by the FlexChartBase class. The property can be set to any of the values from the Orientation enumeration.
- **Position**: Set the legend on top, bottom, left, right, or let it be positioned automatically by using the LegendPosition property that accepts values from the Position enumeration. Setting the Position property to None hides the legend.
- **Styling**: Customize the overall appearance of the legend, such as setting stroke color or changing font by using styling properties accessible through the LegendStyle property. The styling properties Stroke, FontSize, and FontStyle are provided by the ChartStyle class.

Copyright © 2019 GrapeCity, Inc. All rights reserved.

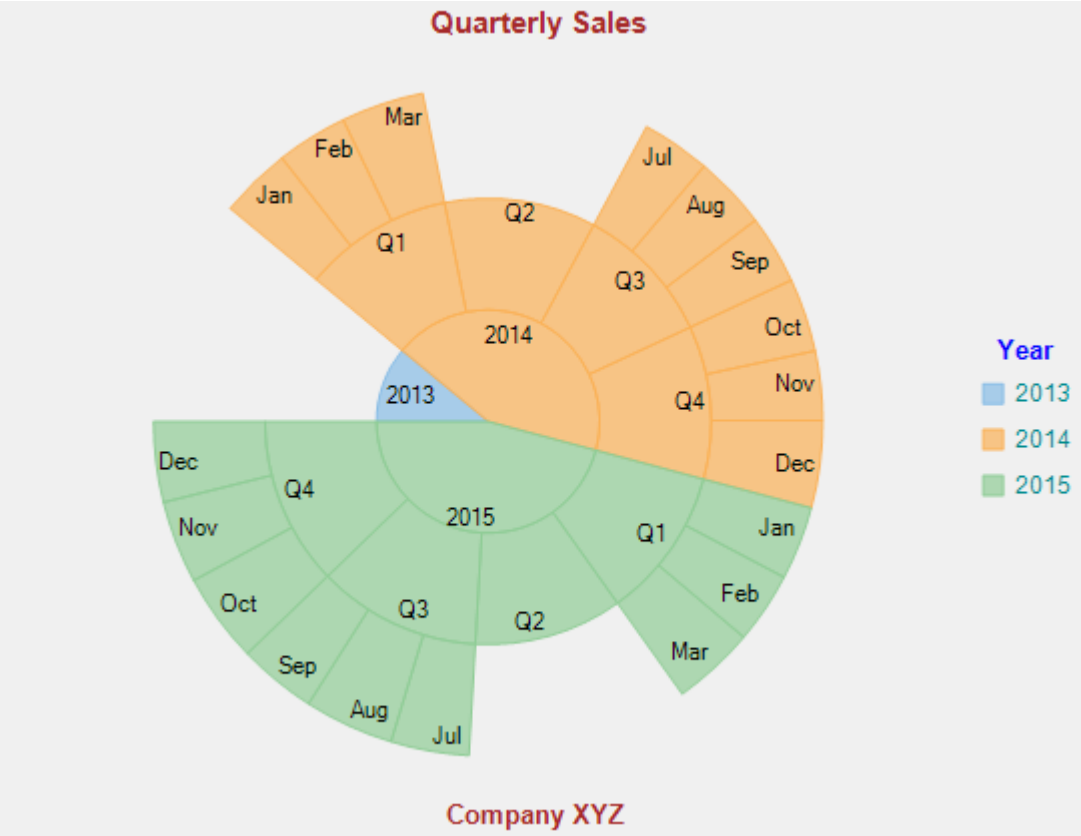
- **Title and title styling**: Specify the legend title using the LegendTitle property that accepts a string. Once you have set the title, you can style it using the LegendTitleStyle property that provides access to the customization properties of the ChartStyle class.

Header and Footer

Header and Footer are descriptive texts at the top and bottom of the chart that provide information about the overall chart data. You can access Header and Footer of Sunburst chart by using the Header and the Footer property respectively, of the FlexChartBase class. Possible customizations with Header and Footer are as follows:

- **Font**: Change the font family, font size, and font style of Header and Footer using the various font properties of the ChartStyle class accessible through the HeaderStyle or FooterStyle property of the FlexChartBase class.
- **Stroke**: Set stroke of the titles for enhanced appeal by using the Stroke property.

The following image displays Sunburst chart with the legend and titles set.



The following code snippet illustrates how to set respective properties for the legend and titles customization. This code uses the sample created in Quick Start.

XAML

```
<c1:C1Sunburst x:Name="sunburst"
               SelectionMode="Point"
               SelectedItemOffset="0.1"
               SelectedItemPosition="Top"
               Header="Quarterly Sales"
               Footer="Company XYZ"
               Reversed="True"
               Palette="Dark"
               Offset="0.1"
               ItemsSource="{Binding HierarchicalData}"
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
                Binding="Value"
                BindingName="Year,Quarter,Month"
                ChildItemsPath="Items"
                ToolTipContent="{}{name}&#x000A;{y}"
                Height="439"
                VerticalAlignment="Bottom"
                LegendOrientation="Vertical"
                LegendTitle="Year" >
    <c1:C1Sunburst.LegendStyle>
        <c1:ChartStyle FontFamily="Arial"
                       FontSize="9"
                       Stroke="DarkCyan"/>
    </c1:C1Sunburst.LegendStyle>
    <c1:C1Sunburst.LegendTitleStyle>
        <c1:ChartStyle FontFamily="Arial"
                       FontSize="10"
                       Stroke="Blue"/>
    </c1:C1Sunburst.LegendTitleStyle>
    <c1:C1Sunburst.HeaderStyle>
        <c1:ChartStyle FontFamily="Arial"
                       FontSize="11"
                       FontWeight="Bold"
                       Stroke="Brown"/>
    </c1:C1Sunburst.HeaderStyle>
    <c1:C1Sunburst.FooterStyle>
        <c1:ChartStyle FontFamily="Arial"
                       FontSize="10"
                       FontWeight="Bold"
                       Stroke="Brown"/>
    </c1:C1Sunburst.FooterStyle>
    <c1:C1Sunburst.DataLabel>
        <c1:PieDataLabel Position="Center"
                         Content="{}{name}" />
    </c1:C1Sunburst.DataLabel>
</c1:C1Sunburst>
```

Code

C# copyCode

```
// set the legend orientation
sunburst.LegendOrientation = C1.Chart.Orientation.Vertical;

// set the legend position
sunburst.LegendPosition = C1.Chart.Position.Auto;

// Set the legend font
sunburst.LegendStyle.FontSize = 9;

// Set the legend font style
sunburst.LegendStyle.FontStyle = FontStyles.Normal;

// set the legend title
sunburst.LegendTitle = "Year";

// set the legend title font
sunburst.LegendTitleStyle.FontSize = 10;
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
// set the legend title font style
sunburst.LegendTitleStyle.FontStyle = FontStyles.Normal;

// set the header
sunburst.Header = "Quarterly Sales";

// set the header font
sunburst.HeaderStyle.FontSize = 11;

// set the header font style
sunburst.HeaderStyle.FontStyle = FontStyles.Normal;

// set the header stroke
sunburst.HeaderStyle.Stroke = Brushes.Brown;

// set the footer
sunburst.Footer = "Company XYZ";

// set the footer font size
sunburst.FooterStyle.FontSize = 10;

// set the footer font style
sunburst.FooterStyle.FontStyle = FontStyles.Normal;

// set the footer stroke
sunburst.FooterStyle.Stroke = Brushes.Brown;
```

VB copyCode

```
' set the legend orientation
sunburst.LegendOrientation = C1.Chart.Orientation.Vertical

' set the legend position
sunburst.LegendPosition = C1.Chart.Position.Auto

' Set the legend font
sunburst.LegendStyle.FontSize = 9

' Set the legend font style
sunburst.LegendStyle.FontStyle = FontStyles.Normal

' set the legend title
sunburst.LegendTitle = "Year"

' set the legend title font
sunburst.LegendTitleStyle.FontSize = 10

' set the legend title font style
sunburst.LegendTitleStyle.FontStyle = FontStyles.Normal

' set the header
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
sunburst.Header = "Quarterly Sales"

' set the header font
sunburst.HeaderStyle.FontSize = 11

' set the header font style
sunburst.HeaderStyle.FontStyle = FontStyles.Normal

' set the header stroke
sunburst.HeaderStyle.Stroke = Brushes.Brown

' set the footer
sunburst.Footer = "Company XYZ"

' set the footer font size
sunburst.FooterStyle.FontSize = 10

' set the footer font style
sunburst.FooterStyle.FontStyle = FontStyles.Normal

' set the footer stroke
sunburst.FooterStyle.Stroke = Brushes.Brown
```

Selection

The Sunburst chart lets you select data points by clicking a Sunburst slice. You can set the SelectionMode property provided by the FlexChartBase class to either of the following values in the ChartSelectionMode enumeration:

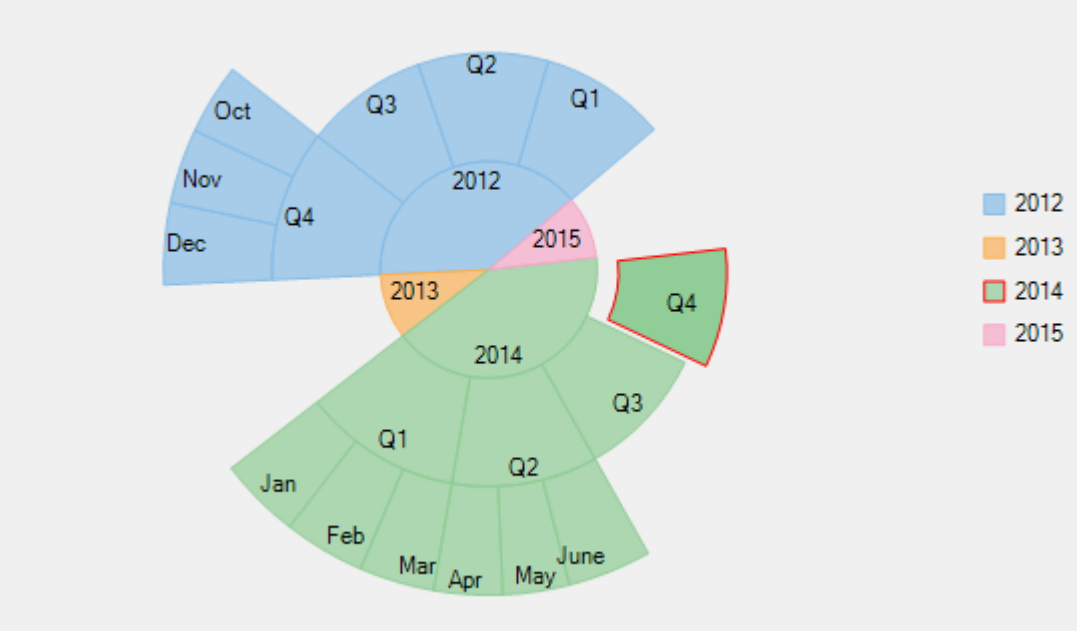
- **None (default)**: Selection is disabled.
- **Point**: A point is selected.

To customize the selection, you can use the SelectedItemOffset and the SelectedItemPosition property provided by the C1FlexPie class. The SelectedItemOffset property enables you to specify the offset of the selected Sunburst slice from the center of the control. And the SelectedItemPosition property allows you to specify the position of the selected Sunburst slice. The SelectedItemPosition property accepts values from the Position enumeration. Setting this property to a value other than 'None' causes the pie to rotate when an item is selected.

In addition, the FlexChartBase class provides the SelectionStyle property that can be used to access the properties provided by the ChartStyle class to style the Sunburst chart.

The following image displays Sunburst chart with a data point selected.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



The following code snippet sets these properties:

XAML

```
<c1:C1Sunburst x:Name="sunburst"
               SelectionMode="Point"
               SelectedItemOffset="0.1"
               SelectedItemPosition="Top"
               Header="Quarterly Sales"
               Footer="Company XYZ"
               Reversed="True"
               Palette="Dark"
               Offset="0.1"
               ItemsSource="{Binding HierarchicalData}"
               Binding="Value"
               BindingName="Year,Quarter,Month"
               ChildItemsPath="Items"
               ToolTipContent="{}{name}&#x000A;{y}"
               Height="439"
               VerticalAlignment="Bottom"
               LegendOrientation="Vertical"
               LegendTitle="Year" >
    <c1:C1Sunburst.LegendStyle>
        <c1:ChartStyle FontFamily="Arial"
                       FontSize="9"
                       Stroke="DarkCyan"/>
    </c1:C1Sunburst.LegendStyle>
    <c1:C1Sunburst.LegendTitleStyle>
        <c1:ChartStyle FontFamily="Arial"
                       FontSize="10"
                       Stroke="Blue"/>
    </c1:C1Sunburst.LegendTitleStyle>
    <c1:C1Sunburst.HeaderStyle>
        <c1:ChartStyle FontFamily="Arial"
                       FontSize="11"
                       FontWeight="Bold"
                       Stroke="Brown"/>
    </c1:C1Sunburst.HeaderStyle>
    <c1:C1Sunburst.FooterStyle>
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
        <c1:ChartStyle FontFamily="Arial"
                       FontSize="10"
                       FontWeight="Bold"
                       Stroke="Brown"/>
    </c1:C1Sunburst.FooterStyle>
    <c1:C1Sunburst.DataLabel>
        <c1:PieDataLabel Position="Center"
                         Content="{}{name}" />
    </c1:C1Sunburst.DataLabel>
</c1:C1Sunburst>
```

Code

C# copyCode

```
// set the SelectionMode property
sunburst.SelectionMode = C1.Chart.ChartSelectionMode.Point;

// set the SelectedItemOffset property
sunburst.SelectedItemOffset = 0.1;

// set the SelectedItemPosition property
sunburst.SelectedItemPosition = C1.Chart.Position.Top;
```

VB copyCode

```
' set the SelectionMode property
sunburst.SelectionMode = C1.Chart.ChartSelectionMode.Point

' set the SelectedItemOffset property
sunburst.SelectedItemOffset = 0.1

' set the SelectedItemPosition property
sunburst.SelectedItemPosition = C1.Chart.Position.Top
```

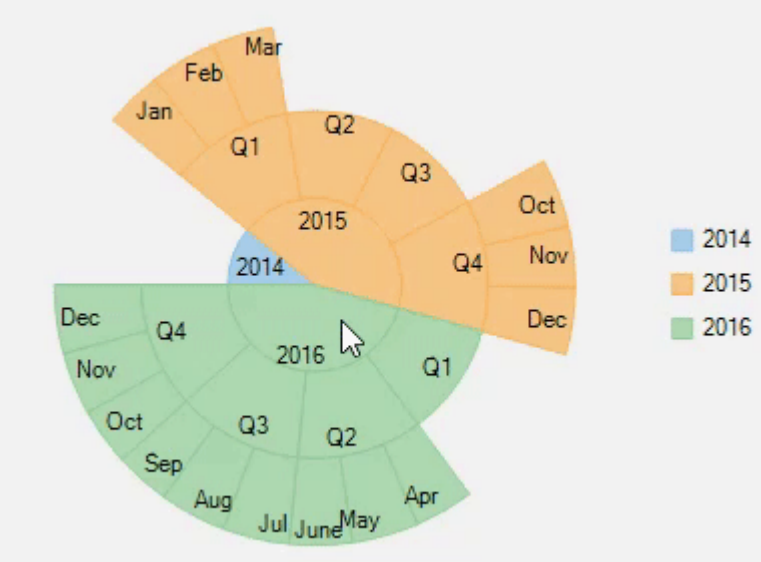
Drilldown

Drilling down data to get in to the details and access lower levels in data hierarchy of Sunburst chart can be quite helpful while analysis. Sunburst chart provides **Drilldown** property to enable the functionality of drilling down and drilling back up the data at run-time.

End users can focus and drill down a data item in a Sunburst chart by simply clicking the desired slice. Whereas, to move up in the hierarchy, users simply need to right-click in the plot area.

The following gif image demonstrates drilling-down by showing data points of the clicked Sunburst slice.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



Note that drill down feature of Sunburst works only when selection of Sunburst slice is disabled, that is, SelectionMode property is set to **None**. For more information on selection, see Selection in Sunburst.

Data Labels

Data labels provide additional information about the data points. These labels make a chart easier to understand because they show details about a slice in the pie.

To understand the working of data labels in Sunburst chart, refer to the following sections.

Adding and Positioning Data Labels
: Learn how to add data labels and set their position on the chart.

Formatting Data Labels
: Learn how to perform styling and formatting of data labels.

Managing Overlapped Data Labels
: Learn how to manage overlapping data labels in Sunburst chart.

Adding and Positioning Labels

With Sunburst chart, you can configure the arrangement and display properties for data labels depending on what suits your needs the best. By default, the data labels are not displayed on the chart, however, you can enable them by setting the Position and Content properties of DataLabel class.

The example code below uses the Position and Content properties to enable data labels and set their position.

XAML

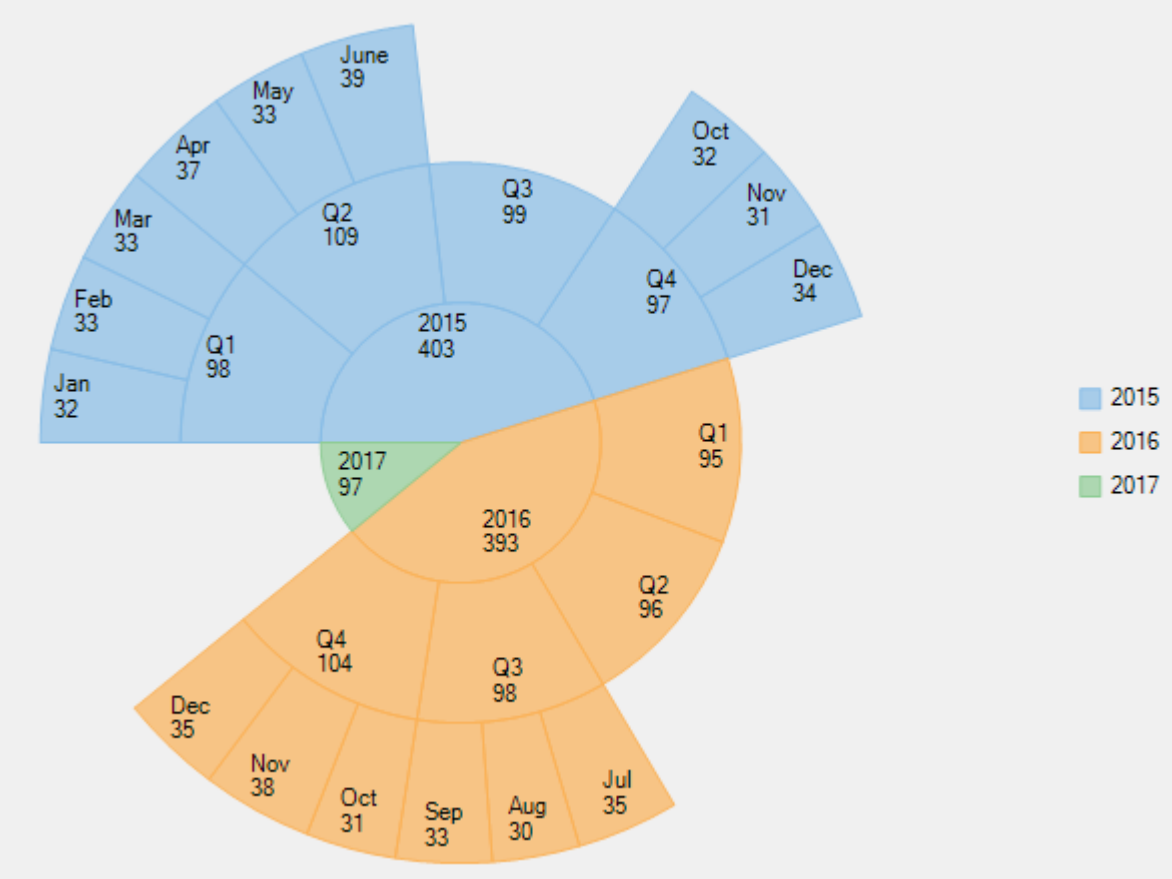
```
<c1:C1Sunburst.DataLabel>
<c1:PieDataLabel Position="Inside"
Content="{}{name}">
</c1:PieDataLabel>
</c1:C1Sunburst.DataLabel>
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Code

HTML

```
sunburst.DataLabel.Content = "{Name}{value}";
sunburst.DataLabel.Position = C1.Chart.PieLabelPosition.Inside;
```



Formatting Data Labels

Sunburst provides various options to format data labels according to your requirements. You can use connecting lines to connect the data labels, set and style borders of data labels, and customize the appearance of data labels.

The topic comprises of three sections:

- **Setting and Styling Borders**
- **Connecting DataLabels to Data Points**
- **Modifying Appearance**

Setting and Styling Borders

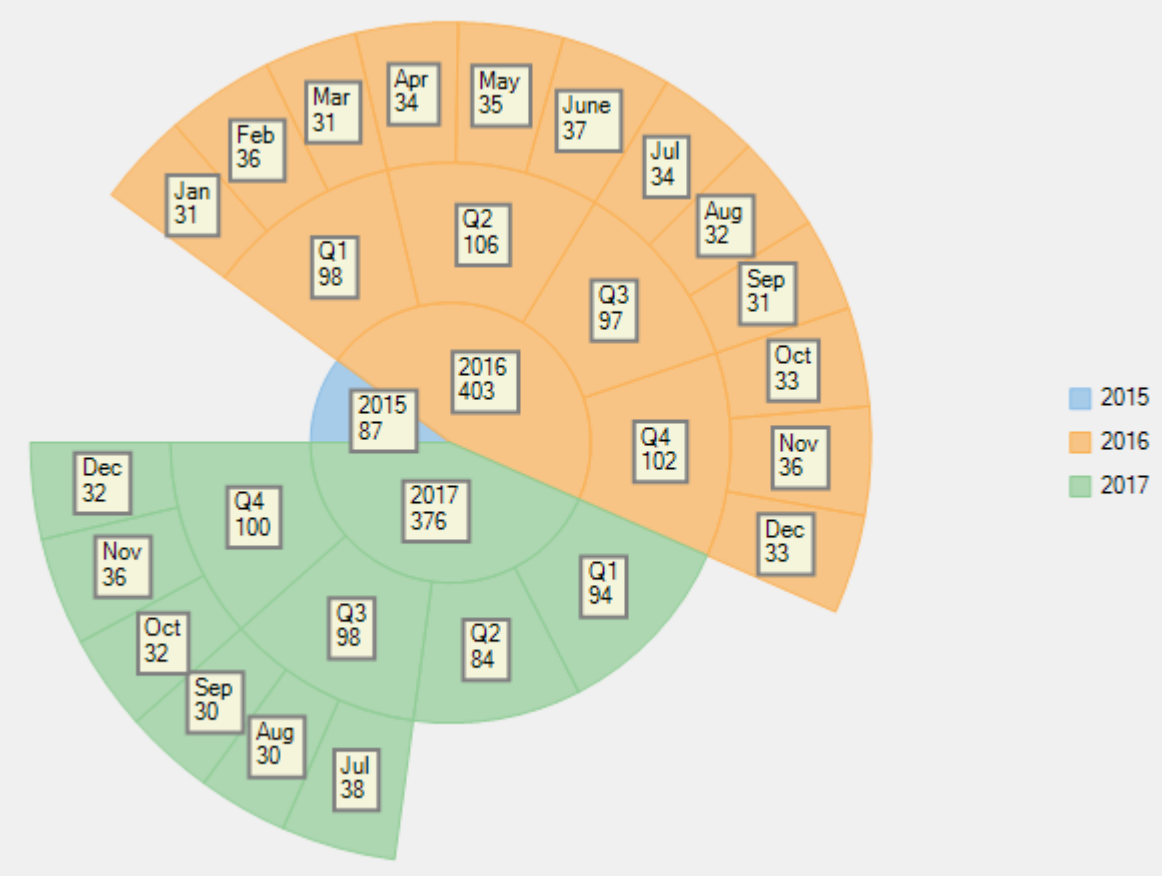
To add and style borders to Sunburst data labels, set the Border and BorderStyle properties provided by DataLabelBase class.

Use the following code snippet to add borders to data labels of Sunburst.

C#

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
// Enable Border
sunburst.DataLabel.Border = true;
sunburst.DataLabel.BorderStyle.StrokeThickness = 2;
```



Connecting DataLabels to Data Points

In case the data labels are placed away from the data points, you can connect them using connecting lines.

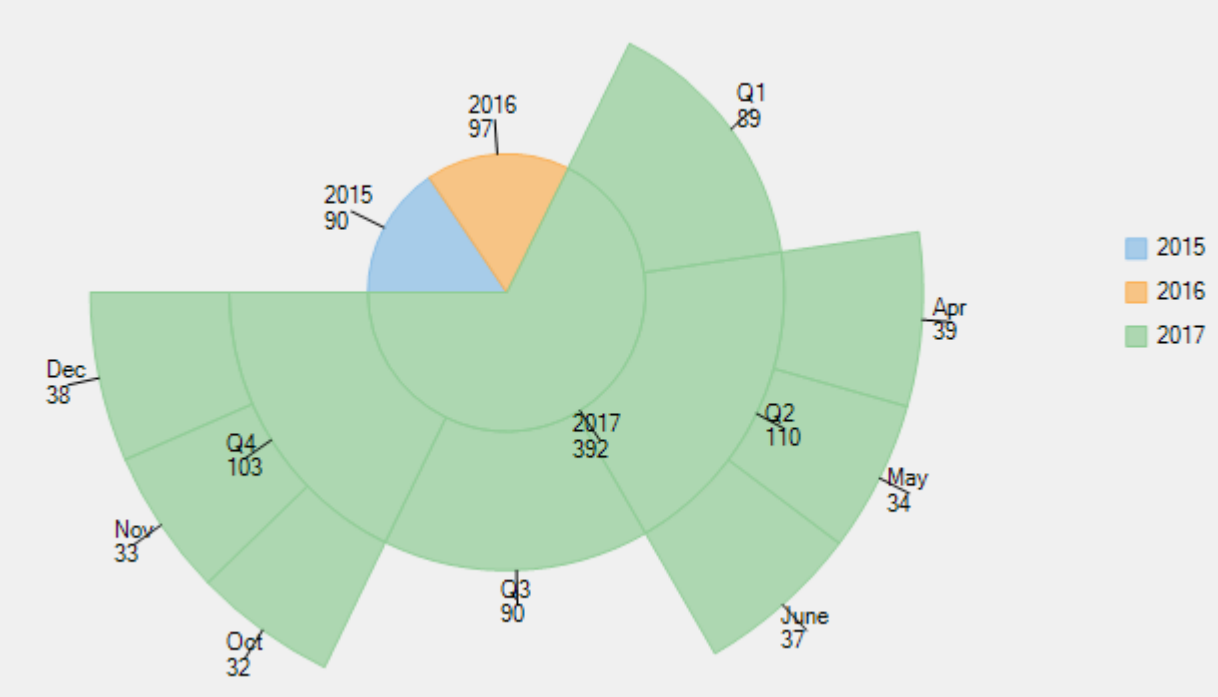
To enable connecting lines in Sunburst chart, you need to use the ConnectingLine property.

Use the following code snippet to set the connecting lines.

C#

```
//Enable connecting line
sunburst.DataLabel.ConnectingLine = true;
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.



Modifying Appearance

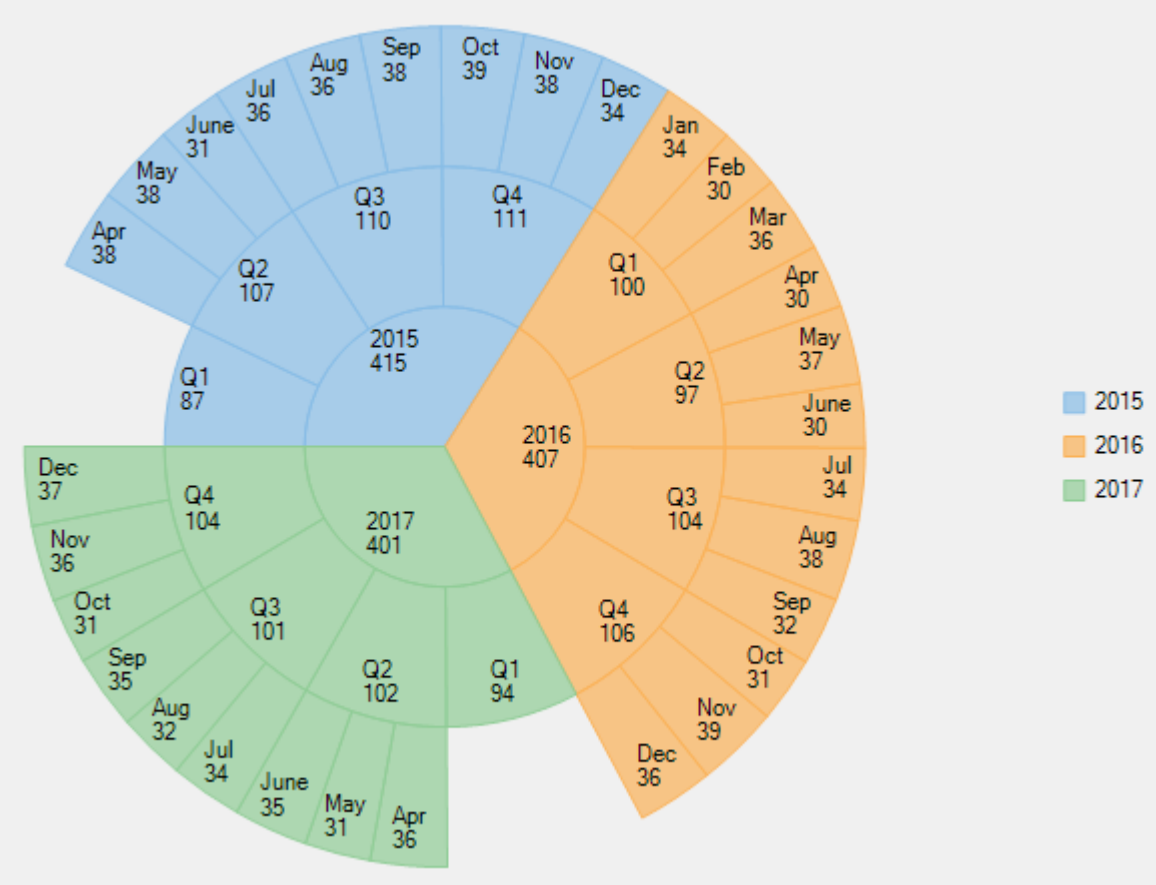
Sunburst includes various styling options, to enhance the clarity of data labels. To modify the appearance of Sunburst chart, you need to use the Style property. It allows you to modify the font family, fill color, use stroke brush for data labels, set width for stroke brush and more.

In the example code, we have modified the font used in the chart and set the stroke width property. Use the following code snippet to modify the appearance of the chart.

C#

```
// Modifying appearance of data labels
sunburst.DataLabel.Style.FontFamily = new
FontFamily("GenericSansSerif");
sunburst.DataLabel.Style.StrokeThickness = 2;
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.



Manage Overlapped Data Labels

A common issue pertaining to charts is overlapping of data labels that represent data points. In most cases, overlapping occurs due to long text in data labels or large numbers of data points.

To manage overlapped data labels in Sunburst chart, you can make use of Overlapping property provided by PieDataLabel class. The Overlapping property accepts the following values from the PieLabelOverlapping enumeration.

Enumeration	Description
PieLabelOverlapping.Default	Show all labels including the overlapping ones.
PieLabelOverlapping.Hide	Hides the overlapping labels, if its content is larger than the corresponding pie segment.
PieLabelOverlapping.Trim	Trim overlapping data labels, if its width is larger than the corresponding pie segment.

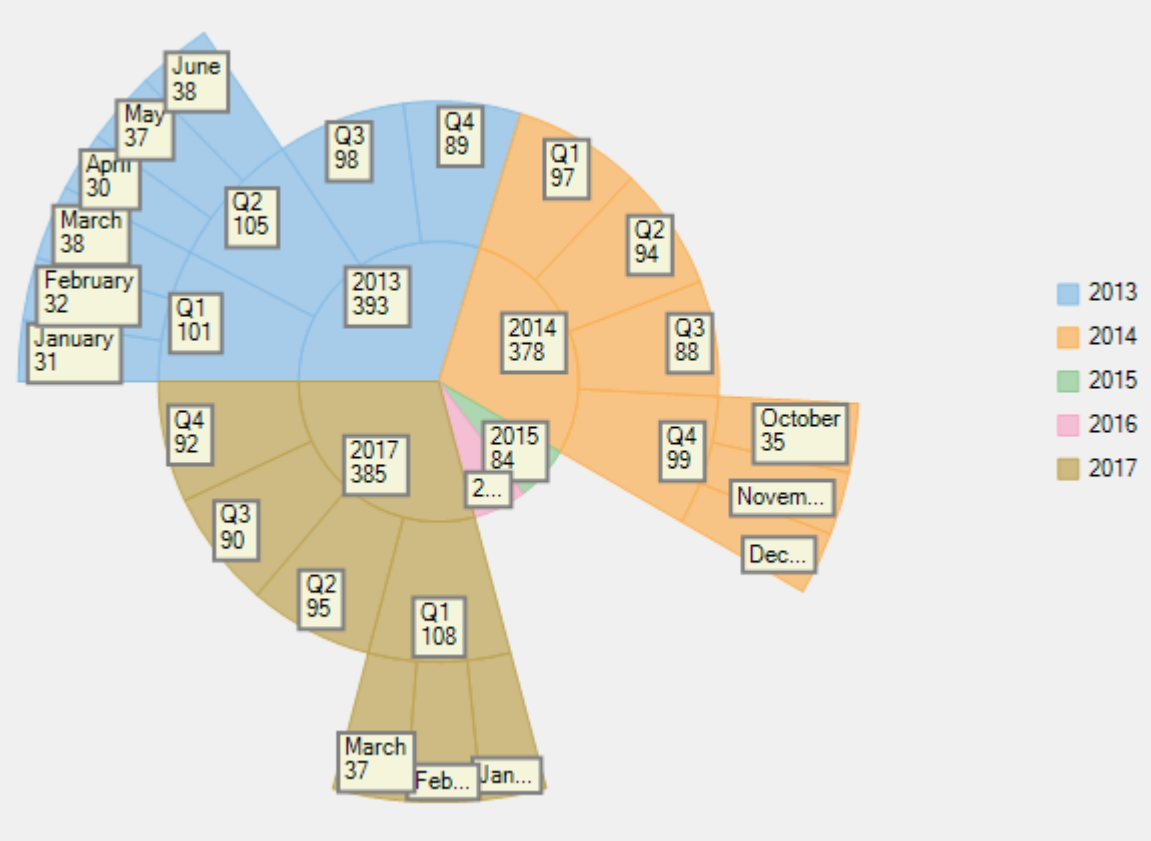
Use the following code to manage overlapping data labels.

C#

```
//Set Overlapping property
sunburst.DataLabel.Overlapping = C1.Chart.PieLabelOverlapping.Trim;
```

The following image shows how Sunburst appears after setting the Overlapping property.

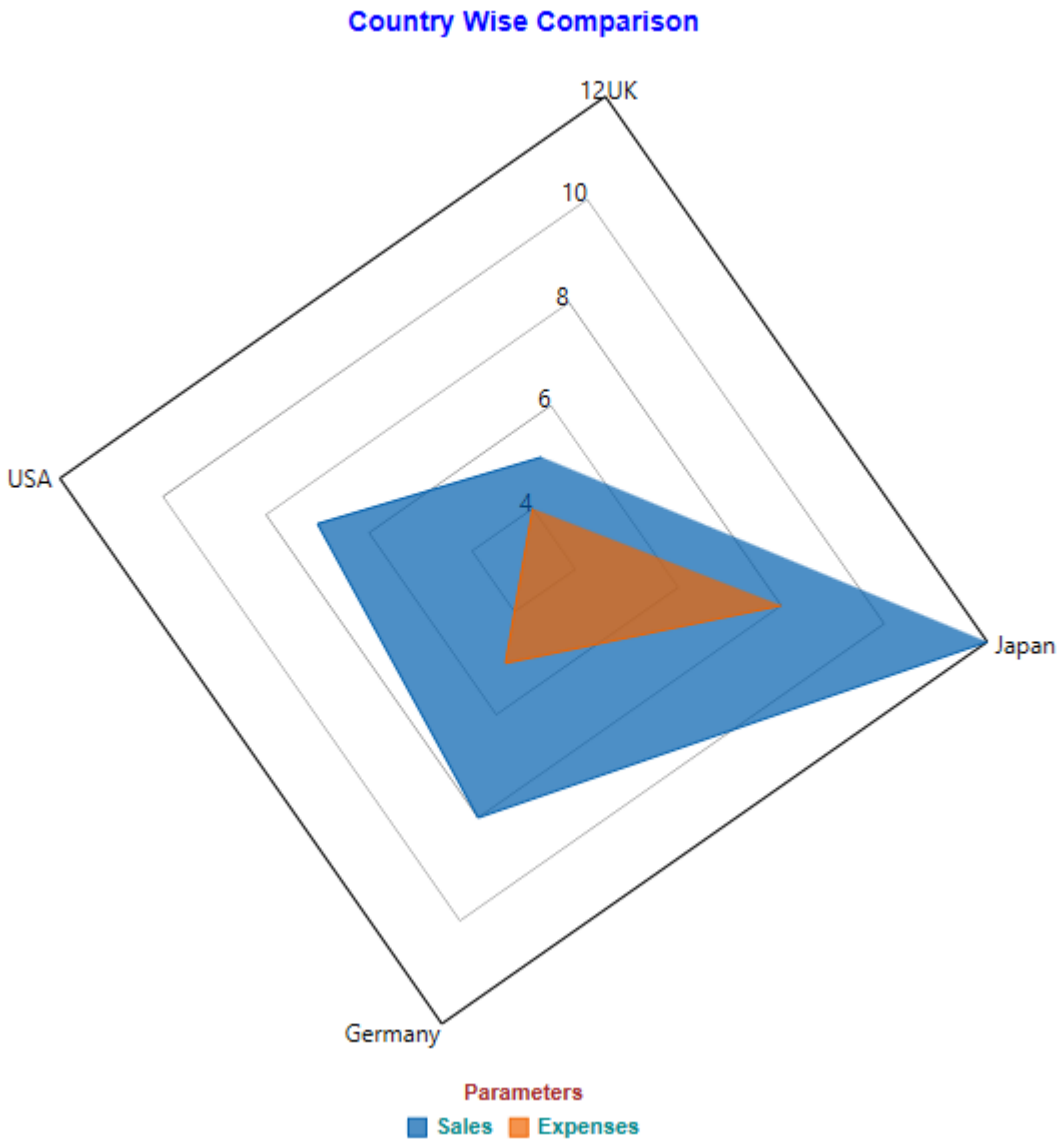
Copyright © 2019 GrapeCity, Inc. All rights reserved.



Copyright © 2019 GrapeCity, Inc. All rights reserved.

FlexRadar

FlexRadar is a radar chart that is also known as polar chart, star chart, web chart, or spider chart due to its appearance. The chart plots value of each category along a separate axis that starts from the center and ends on the outer ring. All axes are arranged radially, with equal distances between each other, while maintaining the same scale between all axes. Each category value is plotted along its individual axis and all the values are connected together to form a polygon. Common business applications of FlexRadar can include skill analysis of employees and product comparison. It is important to note that the FlexRadar control represents a polar chart when X values are numbers that specify angular values in degrees.



To know more about FlexRadar, click the following links:

- Quick Start
- Key Features
- Chart Types
- Legend and Titles

Quick Start

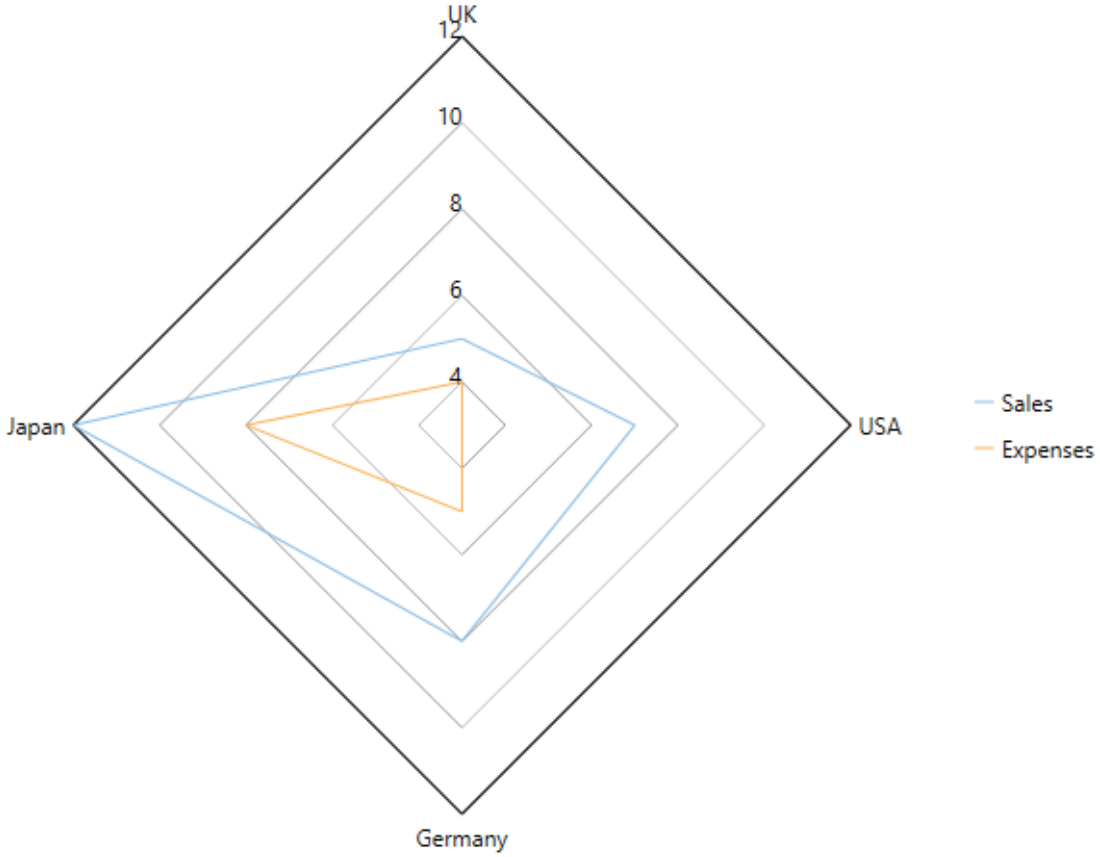
Copyright © 2019 GrapeCity, Inc. All rights reserved.

This quick start is intended to guide you through a step-by-step process of creating a simple FlexRadar application and running the same in Visual Studio.

To quickly get started with FlexRadar and observe how it appears on running the application, follow these steps:

1. **Add FlexRadar to the Application**
2. **Bind FlexRadar to the Data Source**
3. **Run the Application**

The following image displays how a basic FlexRadar appears after completing the steps mentioned above.



Step 1: Add FlexRadar to the Application

1. Create a **WPF Application** in Visual Studio.
2. Drag and drop the **C1FlexRadar** control from the **Toolbox** to the MainWindow.
   The .dll files which are automatically added to the application, are as follows:
   - **C1.WPF.4.dll**
   - **C1.WPF.DX.4.dll**
   - **C1.WPF.FlexChart.4.dll**

   The XAML markup resembles the following code in the **<Grid></Grid>** tags.
   - **XAML**

```
<c1:C1FlexRadar HorizontalAlignment="Left"
                Height="100"
                Margin="0"
                VerticalAlignment="Top"
                Width="100"/>
```

Step 2: Bind FlexRadar to the Data Source

In this step, first create a class DataCreator that generates sales and expenses data for different countries. Next, bind

Copyright © 2019 GrapeCity, Inc. All rights reserved.

FlexRadar to the created data source using the ItemsSource property provided by the FlexChartBase class. Then, specify fields containing X values and Y values for FlexRadar using the BindingX and the Binding property, respectively.

1. Add a class, **DataCreator** and add the following code.
    - **Visual Basic**

```
Class DataCreator
    Public Shared Function CreateData() As List(Of DataItem)
        Dim data = New List(Of DataItem)()
        data.Add(New DataItem("UK", 5, 4))
        data.Add(New DataItem("USA", 7, 3))
        data.Add(New DataItem("Germany", 8, 5))
        data.Add(New DataItem("Japan", 12, 8))
        Return data
    End Function
End Class

Public Class DataItem
    Public Sub New(country__1 As String, sales__2 As Integer, expenses__3 As Integer)
        Country = country__1
        Sales = sales__2
        Expenses = expenses__3
    End Sub

    Public Property Country() As String
        Get
            Return m_Country
        End Get
        Set
            m_Country = Value
        End Set
    End Property
    Private m_Country As String
    Public Property Sales() As Integer
        Get
            Return m_Sales
        End Get
        Set
            m_Sales = Value
        End Set
    End Property
    Private m_Sales As Integer
    Public Property Expenses() As Integer
        Get
            Return m_Expenses
        End Get
        Set
            m_Expenses = Value
        End Set
    End Property
    Private m_Expenses As Integer
End Class
```

    - **C#**

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace FlexRadarQuickStart
{
    class DataCreator
    {
        public static List<DataItem> CreateData()
        {
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
            var data = new List<DataItem>();
            data.Add(new DataItem("UK", 5, 4));
            data.Add(new DataItem("USA", 7, 3));
            data.Add(new DataItem("Germany", 8, 5));
            data.Add(new DataItem("Japan", 12, 8));
            return data;
        }
    }

    public class DataItem
    {
        public DataItem(string country, int sales, int expenses)
        {
            Country = country;
            Sales = sales;
            Expenses = expenses;
        }

        public string Country { get; set; }
        public int Sales { get; set; }
        public int Expenses { get; set; }
    }
}
```

2. Edit the XAML markup to provide data to FlexRadar.

    - **XAML**

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:FlexRadarQuickStart"
    xmlns:Chart="clr-namespace:C1.WPF.Chart;assembly=C1.WPF.FlexChart.4"
    x:Class="FlexRadarQuickStart.MainWindow"
    mc:Ignorable="d"
    DataContext = "{Binding RelativeSource={RelativeSource Mode=Self}}">

    <Grid>
        <Chart:C1FlexRadar ItemsSource="{Binding DataContext.Data}"
                           BindingX="Country"
                           Margin="84,50,216,142">
            <Chart:Series SeriesName="Sales"
                          Binding="Sales"/>
            <Chart:Series SeriesName="Expenses"
                          Binding="Expenses" />
        </Chart:C1FlexRadar>
    </Grid>

</Window>
```

3. Switch to **Code view** and add the following code.

    - **Visual Basic**

```
Partial Public Class MainWindow
    Inherits Window
    Private _data As List(Of DataItem)
    Public Sub New()
        Me.InitializeComponent()
        Dim flexradar As New C1.WPF.Chart.C1FlexRadar()
    End Sub

    Public ReadOnly Property Data() As List(Of DataItem)
        Get
            If _data Is Nothing Then
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
                _data = DataCreator.CreateData()
            End If

            Return _data
        End Get
    End Property
End Class
```

- **C#**

```
using System.Collections.Generic;
using System.Windows;

namespace FlexRadarQuickStart
{
    public partial class MainWindow : Window
    {
        private List<DataItem> _data;
        public MainWindow()
        {
            this.InitializeComponent();
        }

        public List<DataItem> Data
        {
            get
            {
                if (_data == null)
                {
                    _data = DataCreator.CreateData();
                }

                return _data;
            }
        }
    }
}
```

Step 3: Run the Application

Press F5 to run the application and observe the output.

Key Features

Some of the key features of FlexRadar are as follows:

- **Reversed FlexRadar**: Create a reversed FlexRadar by setting the Reversed property to True. A reversed FlexRadar is the one in which direction of plotting is reversed.
- **Start angle:** Set the start angle of FlexRadar by setting the StartAngle property provided by the C1FlexRadar class to double type values. A start angle is the angle that is set in degrees to start drawing radial axes in the clockwise direction.
- **Header and Footer**: Use simple properties to set and customize Header and Footer of FlexRadar. For more information, refer to Legend and Titles.
- **Legend**: Perform various customizations including setting orientation, position, or styling the legend of FlexRadar. For more information, refer to Legend and Titles
- **Chart types:** Visualize data using different chart types within FlexRadar. For more information, refer to Chart Types.

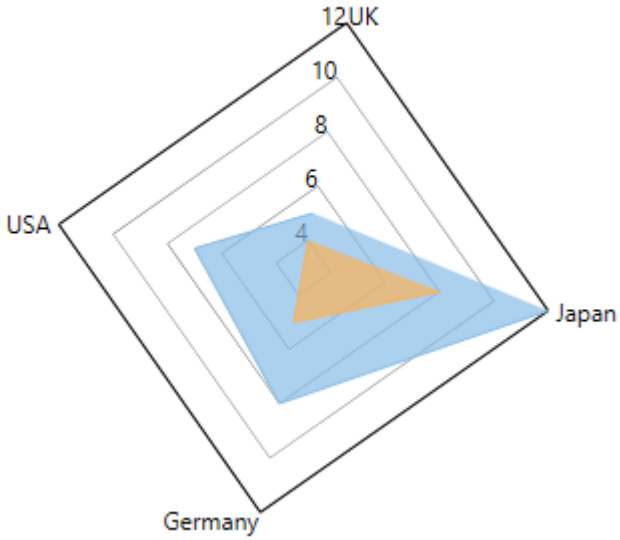
Chart Types

Copyright © 2019 GrapeCity, Inc. All rights reserved.

FlexRadar allows you to work with different chart types to meet your data visualization needs. You can use chart types from Area to Scatter to display areas filled with colors or patterns within data depicted within FlexRadar. To set different chart types for FlexRadar, set the ChartType property of C1FlexRadar to any of the following values in the RadarChartType enumeration:

- **Area**: Shows area below the line filled with color.
- **Line**: Shows trends over a period of time or across categories.
- **LineSymbols**: Shows line chart with a symbol on each data point.
- **Scatter**: Shows patterns within data using X and Y coordinates.

The following image displays FlexRadar with the chart type as Area.



The following code snippet sets the ChartType property in code using the sample created in Quick Start.

XAML

```
<Chart:C1FlexRadar Header="Country Wise Comparison"
                   LegendOrientation="Vertical"
                   LegendPosition="Right"
                   LegendTitle="Parameters"
                   ChartType="Area"
                   Reversed="True"
                   StartAngle="10"
                   ItemsSource="{Binding DataContext.Data}"
                   BindingX="Country"
                   Margin="54,78,58,10" Palette="Standard">
    <Chart:C1FlexRadar.HeaderStyle>
        <Chart:ChartStyle FontFamily="Arial"
                          FontSize="14"
                          FontWeight="Bold"
                          Stroke="Blue"/>
    </Chart:C1FlexRadar.HeaderStyle>
    <Chart:C1FlexRadar.FooterStyle>
        <Chart:ChartStyle FontFamily="Arial"
                          FontSize="12"
                          FontWeight="Bold"/>
    </Chart:C1FlexRadar.FooterStyle>
    <Chart:C1FlexRadar.LegendStyle>
        <Chart:ChartStyle FontFamily="Arial"
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
                            FontSize="11"
                            FontWeight="Bold"
                            Stroke="DarkCyan"/>
    </Chart:C1FlexRadar.LegendStyle>
    <Chart:C1FlexRadar.LegendTitleStyle>
        <Chart:ChartStyle FontFamily="Arial"
                            FontSize="11"
                            FontWeight="Bold"
                            Stroke="Brown"/>
    </Chart:C1FlexRadar.LegendTitleStyle>
    <Chart:Series SeriesName="Sales"
                  Binding="Sales"/>
    <Chart:Series SeriesName="Expenses"
                  Binding="Expenses" />
</Chart:C1FlexRadar>
```

Code

C# copyCode

```
// set the FlexRadar chart type
flexRadar.ChartType = C1.Chart.RadarChartType.Area;
```

VB copyCode

```
' set the FlexRadar chart type
flexRadar.ChartType = C1.Chart.RadarChartType.Area
```

Legend and Titles

Legend

The legend displays entries for series with their names and predefined symbols. In FlexRadar, you can perform various customizations with the legend of FlexRadar, as follows:

- **Orientation**: Set the orientation of the legend as horizontal, vertical, or automatic by using the LegendOrientation property provided by the FlexChartBase class. The property can be set to any of the values from the Orientation enumeration.
- **Position**: Set the legend on top, bottom, left, right, or let it be positioned automatically by using the LegendPosition property that accepts values from the Position enumeration. Setting the LegendPosition property to None hides the legend.
- **Styling**: Customize the overall appearance of the legend, such as setting stroke color or changing font by using styling properties accessible through the LegendStyle property. The styling properties Stroke, FontSize, and FontStyle are provided by the ChartStyle class.
- **Title and title styling**: Specify the legend title using the LegendTitle property that accepts a string. Once you have set the title, you can style it using the LegendTitleStyle property that provides access to the customization properties of the ChartStyle class.

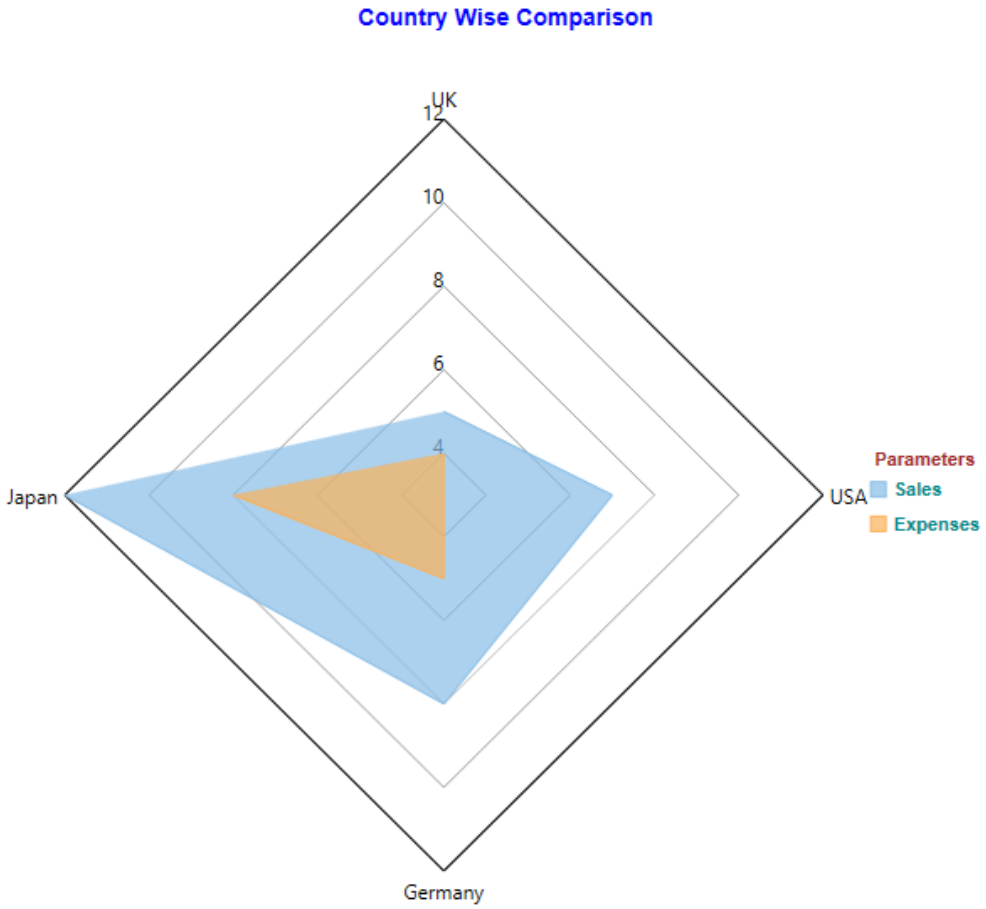
Header and Footer

Header and Footer are descriptive texts at the top and bottom of the chart that provide information about the overall chart data. You can access Header and Footer of FlexRadar by using the Header and the Footer property respectively, of the FlexChartBase class. Possible customizations with Header and Footer are as follows:

Copyright © 2019 GrapeCity, Inc. All rights reserved.

- **Font**: Change the font family, font size, and font style of Header and Footer using the various font properties of the ChartStyle class accessible through the HeaderStyle or FooterStyle property of the FlexChartBase class.
- **Stroke**: Set stroke of the titles for enhanced appeal by using the Stroke property.

The following image displays FlexRadar with the legend and titles set.



The following code snippet demonstrates how to set various properties.

XAML

```
<Chart:C1FlexRadar Header="Country Wise Comparison"
                   LegendOrientation="Vertical"
                   LegendPosition="Right"
                   LegendTitle="Parameters"
                   ChartType="Area"
                   Reversed="True"
                   StartAngle="10"
                   ItemsSource="{Binding DataContext.Data}"
                   BindingX="Country"
                   Margin="54,78,58,10" Palette="Standard">
    <Chart:C1FlexRadar.HeaderStyle>
        <Chart:ChartStyle FontFamily="Arial"
                          FontSize="14"
                          FontWeight="Bold"
                          Stroke="Blue"/>
    </Chart:C1FlexRadar.HeaderStyle>
    <Chart:C1FlexRadar.FooterStyle>
        <Chart:ChartStyle FontFamily="Arial"
                          FontSize="12"
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
                        FontWeight="Bold"/>
    </Chart:C1FlexRadar.FooterStyle>
    <Chart:C1FlexRadar.LegendStyle>
        <Chart:ChartStyle FontFamily="Arial"
                        FontSize="11"
                        FontWeight="Bold"
                        Stroke="DarkCyan"/>
    </Chart:C1FlexRadar.LegendStyle>
    <Chart:C1FlexRadar.LegendTitleStyle>
        <Chart:ChartStyle FontFamily="Arial"
                        FontSize="11"
                        FontWeight="Bold"
                        Stroke="Brown"/>
    </Chart:C1FlexRadar.LegendTitleStyle>
    <Chart:Series SeriesName="Sales"
                  Binding="Sales"/>
    <Chart:Series SeriesName="Expenses"
                  Binding="Expenses" />
</Chart:C1FlexRadar>
```

Code

C# copyCode

```
// set the FlexRadar header
flexRadar.Header = "Country Wise Comparison";

// set the legend orientation
flexRadar.LegendOrientation = C1.Chart.Orientation.Horizontal;

// set the legend position
flexRadar.LegendPosition = C1.Chart.Position.Bottom;
```

VB copyCode

```
' set the FlexRadar header
flexRadar.Header = "Country Wise Comparison"

' set the legend orientation
flexRadar.LegendOrientation = C1.Chart.Orientation.Horizontal

' set the legend position
flexRadar.LegendPosition = C1.Chart.Position.Bottom
```

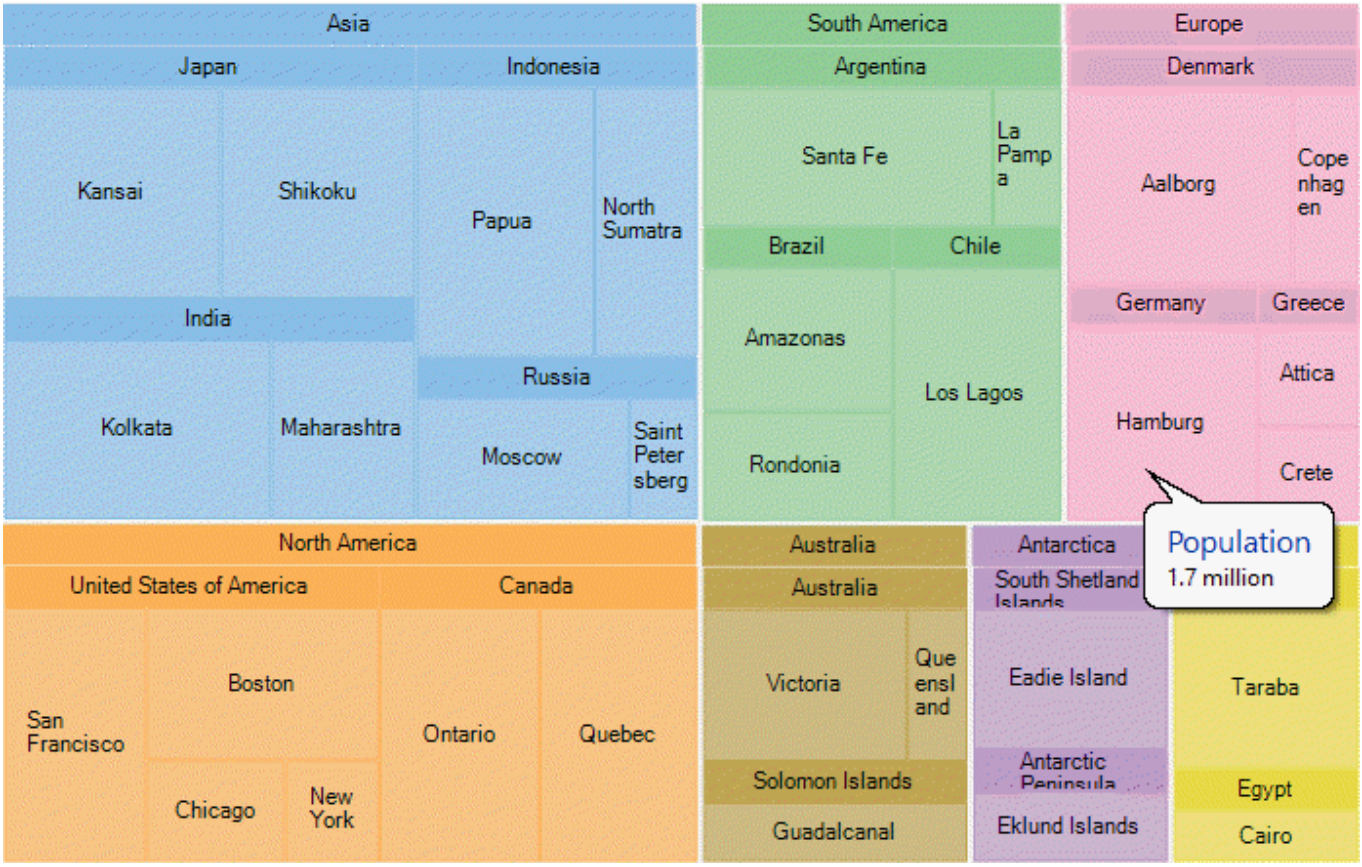
Copyright © 2019 GrapeCity, Inc. All rights reserved.

TreeMap

Hierarchical information and data are useful in varied walks of life and setups, be it family tree, programming, organization structure, or directories. Visualizing such data and spotting information in them is a difficult task, especially if the data is huge. Treemap charts enable visualization of hierarchical data as nested rectangles on a limited space. They are useful in having a quick glimpse of patterns in large data and in comparing proportions.

The **TreeMap** chart control supports binding to data to show hierarchy, and allows user to drill down the data further to numerous levels for detailed analysis. The control can be customized to display data in horizontal, vertical, and squarified layouts of constituting rectangles.

Both TreeMap and Sunburst charts are ideal for displaying and visualizing hierarchical data, but treemap is preferred when space is a constraint as it can show humongous data in a limited area.



The following topics help you get accustomed with the TreeMap control, and explore its advanced capabilities.

Key Features

TreeMap control has numerous features to enable users display hierarchical data in a limited area, and analyze data by comparing the size of tree nodes (or nested rectangles). These are as follows:

- **Hierarchical representation of data**
  TreeMap control is an ideal tool to help users visualize and compare proportions in data values within a hierarchy.
- **Layout**
  TreeMap supports multiple display arrangements, where the tree branches can be shown as squares, horizontal rectangles or vertical rectangles.

Copyright © 2019 GrapeCity, Inc. All rights reserved.

- **Customizable hierarchical levels**
  TreeMap control enables users to vary the depth of data to be visualized and further drill down (or reverse drill down) the data for analysis and comparison.
- **Customizable appearance**
  TreeMap enables users to stylize the control and vary its appearance as per their preference. A set of varied color palettes are available to clearly display categories in a tree map chart.
- **Data binding**
  TreeMap can be bound to different data sources that contain data of varied size, allowing you to display such a data in limited rectangular area.
- **Selection**
  TreeMap enables selecting tree nodes and group of nodes to draw focus on specific data items in the hierarchical data.
- **Optimum space utilization**
  TreeMap is ideal for compact display and visualization of huge data. The nested rectangles and groups constituting the treemap chart adjust their size to fit the display area.

- **Drill down the hierarchy**
  TreeMap allows detailed analysis of hierarchical data by drilling down the data points on a mouse click. This is enabled by specifying the maximum depth of TreeMap hierarchy.

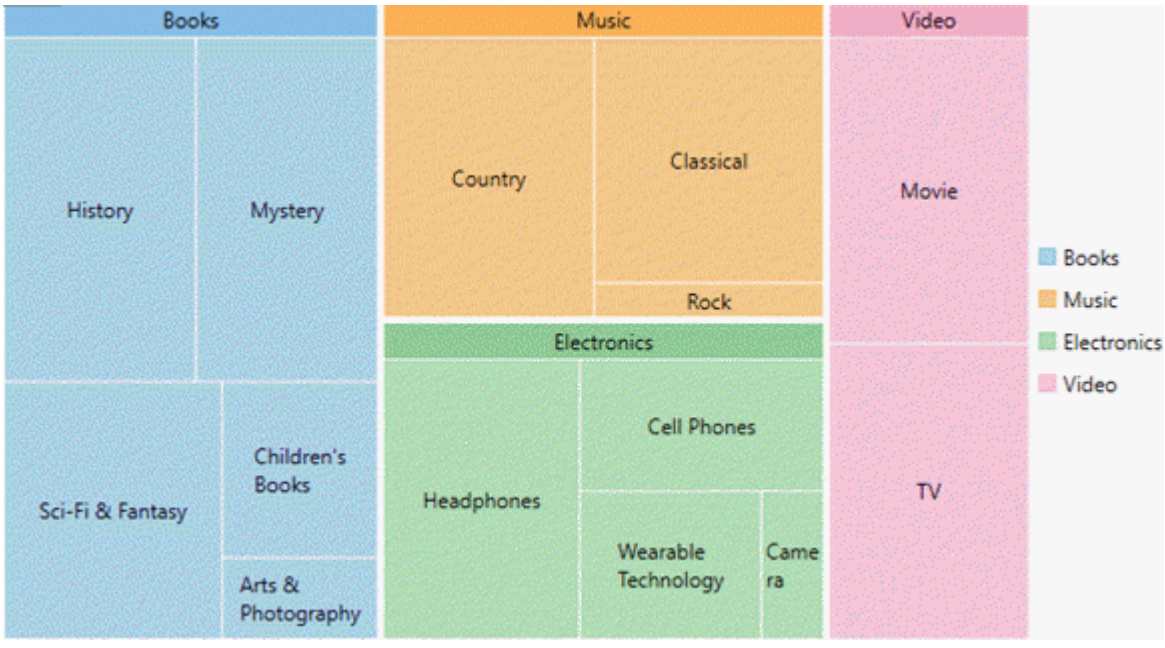
Quick Start

This quick start topic provides step-by-step instructions for adding a TreeMap chart to WPF application, and show hierarchical data in it. In this topic, we consider an example where user wants to compare sales of books, music, videos, and gadgets (like computers and tablets) in XYZ city in a particular year.

The steps to display hierarchical data in TreeMap control are as follows:

- **Step 1: Add TreeMap to project**
- **Step 2: Create a hierarchical data source**
- **Step 3: Bind the TreeMap to data source**
- **Step 4: Build and run the project**

The following image exhibits and compares sales of different varieties of books, music, videos, and gadgets (like computers and tablets) in XYZ city in a particular year.



Copyright © 2019 GrapeCity, Inc. All rights reserved.

Back to Top

You need to set the ItemsSource property, of FlexChartBase class, to point to the collection of objects that contain data points to be plotted on the chart. To generate data items and display them in TreeMap chart, set BindingName and Binding properties. Set the **BindingName** property to the string value that specifies the name of the data item to display as chart rectangles, and **Binding** property to the string value that specifies the name of property of chart items that contain chart values (numeric values that help calculate the size of tree nodes).

To specify the level of hierarchical items to drill down and display in the chart set the MaxDepth property. Also, the display layout of the TreeMap is specified through its ChartType property. Additionally, color palette can be used to stylize the control and change its appearance.

Step 1: Add TreeMap to project

1. Create a **Windows** application in Visual Studio.
2. Drag and drop **C1TreeMap** control from Toolbox to page.
   The following DLLs get added to your application:
   - **C1.WPF.dll**
   - **C1.WPF.FlexChart.dll**

Back to Top

Step 2: Create a hierarchical data source

Switch to the code view and add the following code to generate sales data of Books, Music, Electronic items, Videos, and Computers and tablets.

- **Visual Basic**

```
Private rnd As New Random()
Private Function rand() As Integer
    Return rnd.[Next](10, 100)
End Function

Public ReadOnly Property Data() As Object()
    Get
        Dim data__1 = New Object() {New With {
         .type = "Music",
         .items = New () {New With {
             .type = "Country",
             .items = New () {New With {
                 .type = "Classic Country",
                 .sales = rand()
            }}
        }, New With {
             .type = "Rock",
             .items = New () {New With {
                 .type = "Funk Rock",
                 .sales = rand()
            }}
        }, New With {
             .type = "Classical",
             .items = New () {New With {
                 .type = "Symphonies",
                 .sales = rand()
            }}
        }}
    }, New With {
         .type = "Books",
         .items = New () {New With {
             .type = "Arts & Photography",
             .items = New () {New With {
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
                .type = "Architecture",
                .sales = rand()
            }}
        }, New With {
            .type = "Children's Books",
            .items = New () {New With {
                .type = "Beginning Readers",
                .sales = rand()
            }}
        }, New With {
            .type = "History",
            .items = New () {New With {
                .type = "Ancient",
                .sales = rand()
            }}
        }, New With {
            .type = "Mystery",
            .items = New () {New With {
                .type = "Thriller & Suspense",
                .sales = rand()
            }}
        }, New With {
            .type = "Sci-Fi & Fantasy",
            .items = New () {New With {
                .type = "Fantasy",
                .sales = rand()
            }}
        }}
    }, New With {
        .type = "Electronics",
        .items = New () {New With {
            .type = "Wearable Technology",
            .items = New () {New With {
                .type = "Activity Trackers",
                .sales = rand()
            }}
        }, New With {
            .type = "Cell Phones",
            .items = New () {New With {
                .type = "Accessories",
                .sales = rand()
            }}
        }, New With {
            .type = "Headphones",
            .items = New () {New With {
                .type = "Earbud headphones",
                .sales = rand()
            }}
        }, New With {
            .type = "Camera",
            .items = New () {New With {
                .type = "Digital Cameras",
                .sales = rand()
            }}
        }}
    }, New With {
        .type = "Video",
        .items = New () {New With {
            .type = "Movie",
            .items = New () {New With {
                .type = "Children & Family",
                .sales = rand()
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
            }}
        }, New With {
             .type = "TV",
             .items = New () {New With {
                 .type = "Comedy",
                 .sales = rand()
            }}
        }}
    }}
        Return data__1
    End Get
End Property
```

- **C#**

```
    Random rnd = new Random();
    int rand()
    {
        return rnd.Next(10, 100);
    }

    public object[] Data
    {
        get
        {
            var data = new object[] { new {
        type = "Music",
        items = new [] { new {
            type = "Country",
            items= new [] { new {
                type= "Classic Country",
                sales = rand()
            }}
        }, new {
            type= "Rock",
            items= new [] { new {
                type= "Funk Rock",
                sales= rand()
             } }
        }, new {
            type= "Classical",
            items= new [] { new {
                type= "Symphonies",
                sales= rand()
                } }
  }}
}, new {
        type= "Books",
  items= new [] { new {
            type= "Arts & Photography",
    items= new [] { new {
                type= "Architecture",
      sales= rand()
    }}
  }, new {
            type= "Children's Books",
    items= new [] { new {
                type= "Beginning Readers",
      sales= rand()
    } }
  }, new {
            type= "History",
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
    items= new [] { new {
                type= "Ancient",
      sales= rand()
    } }
  }, new {
             type= "Mystery",
    items= new [] { new {
                type= "Thriller & Suspense",
      sales= rand()
              } }
  }, new {
             type= "Sci-Fi & Fantasy",
    items= new [] { new {
                type= "Fantasy",
      sales= rand()
    }}
  } }
}, new {
        type= "Electronics",
  items= new [] { new {
             type= "Wearable Technology",
    items= new [] { new {
                type= "Activity Trackers",
      sales= rand()
    }}
  }, new {
             type= "Cell Phones",
    items= new [] { new {
                type= "Accessories",
      sales= rand()
    } }
  }, new {
             type= "Headphones",
    items= new [] { new {
                type= "Earbud headphones",
      sales= rand()
    } }
  }, new {
             type= "Camera",
    items= new [] { new {
                type= "Digital Cameras",
      sales= rand()
              } }
  } }
}, new {
        type= "Video",
  items= new [] { new {
             type= "Movie",
    items= new [] { new {
                type= "Children & Family",
      sales= rand()
    } }
  }, new {
             type= "TV",
    items= new [] { new {
                type= "Comedy",
      sales= rand()
    } }
  } }
} };
             return data;
        }
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

Back to Top

Step 3: Bind the TreeMap to data source

To bind the TreeMap control to the data source, use the following code.

XAML copyCode

```
    <Window
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:WpfTreeMapCS"
        xmlns:c1="http://schemas.componentone.com/winfx/2006/xaml"
    x:Class="WpfTreeMapCS.QuickStart"
    DataContext="{Binding RelativeSource={RelativeSource Mode=Self}}"
        mc:Ignorable="d"
        Title="QuickStart"
    Height="312" Width="434">
    <Grid Margin="10,0,0,0">
        <c1:C1TreeMap Binding="sales"
                      MaxDepth="2"
                      BindingName="type"
                      ChildItemsPath="items"
                      ItemsSource="{Binding DataContext.Data}" >
            <c1:C1TreeMap.DataLabel>
                <c1:DataLabel Content="{}{type}"
                              Position="Center">
                    <c1:DataLabel.Style>
                        <c1:ChartStyle/>
                    </c1:DataLabel.Style>
                </c1:DataLabel>
            </c1:C1TreeMap.DataLabel>
        </c1:C1TreeMap>
    </Grid>
</Window>
```

Back to Top

Step 4: Build and run the project

1. Click **Build | Build Solution** to build the project.
2. Press **F5** to run the project.

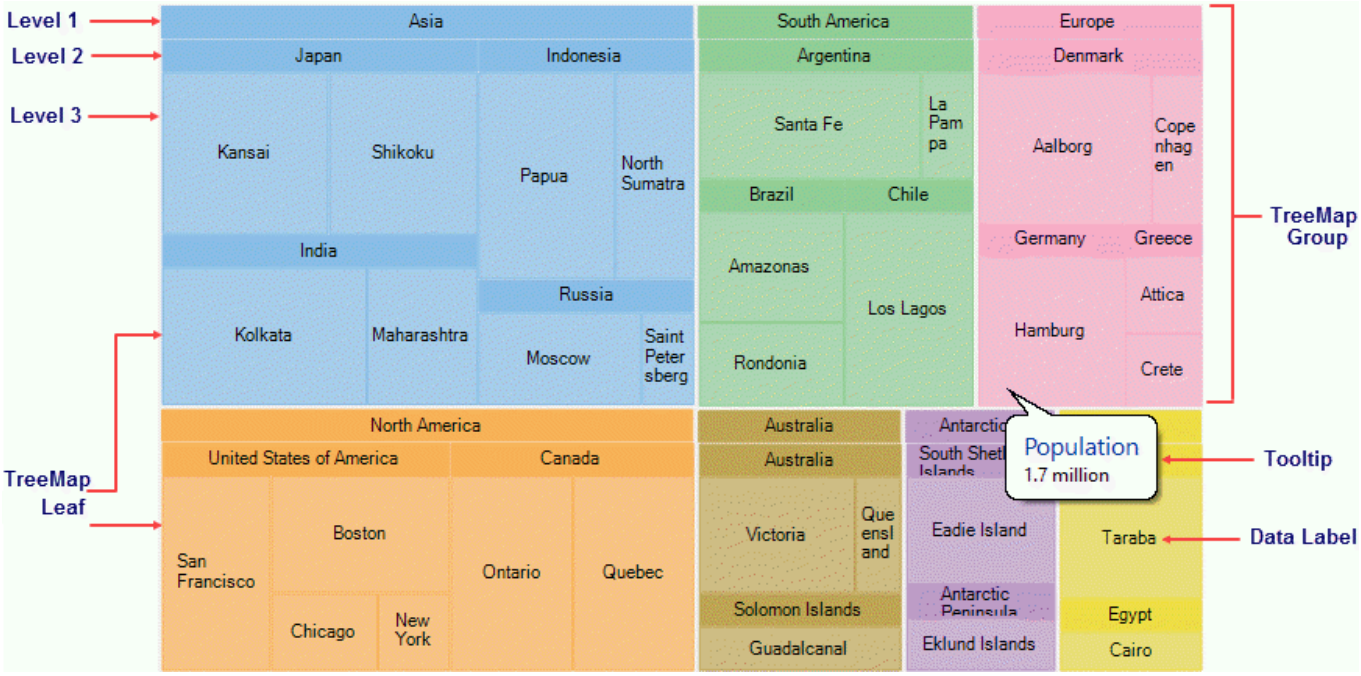
Back to Top

Elements

A tree map chart is composed of rectangles, representing individual data items, which are grouped into categories, to represent the hierarchical nature of data. The individual data items which make group are known as leaf nodes. The size of these nodes are proportional to the data they represent

The following image exhibits main elements of TreeMap Control.

Copyright © 2019 GrapeCity, Inc. All rights reserved.

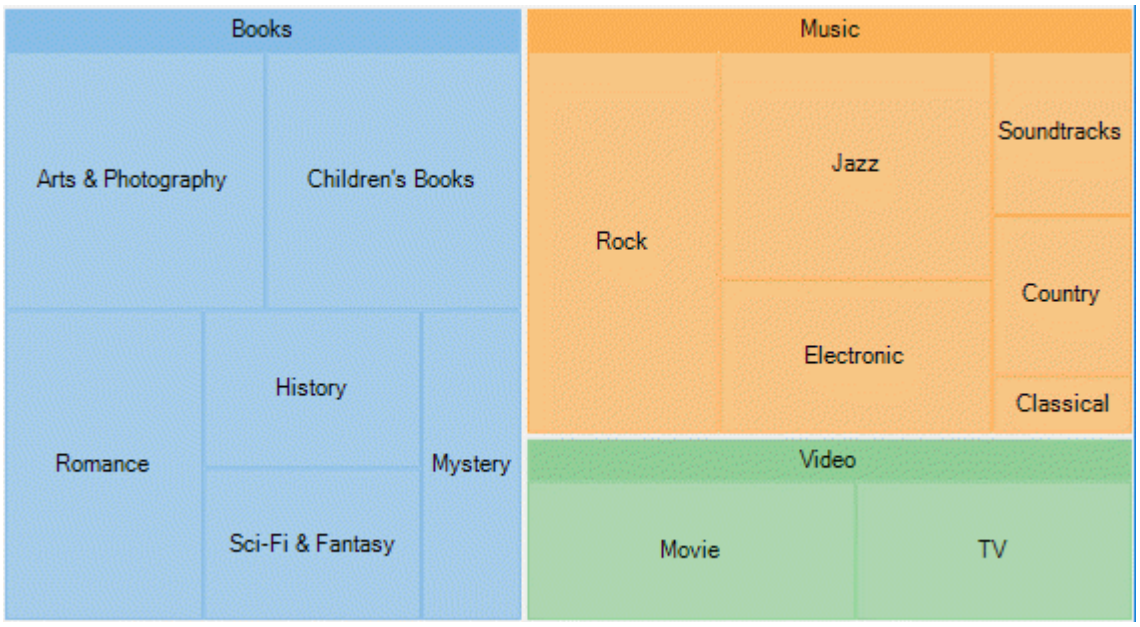


Layouts

TreeMap enables its data items and groups, represented as rectangles, to be displayed in a variety of arrangements. The tree map rectangles can be arranged into squarified, horizontal, and vertical layouts. To set the desired tree map layout, you need to use ChartType property of TreeMap class, which takes TreeMapType enum. The default layout of TreeMap chart control is squarified.

Squarified

The squarified layout tries to arrange the tree map rectangles (data items and groups) as approximate squares. This layout makes it easier to make comparisons and point patterns, as the accuracy of presentation is enhanced in squarified arrangement. This layout is very useful for large data sets.

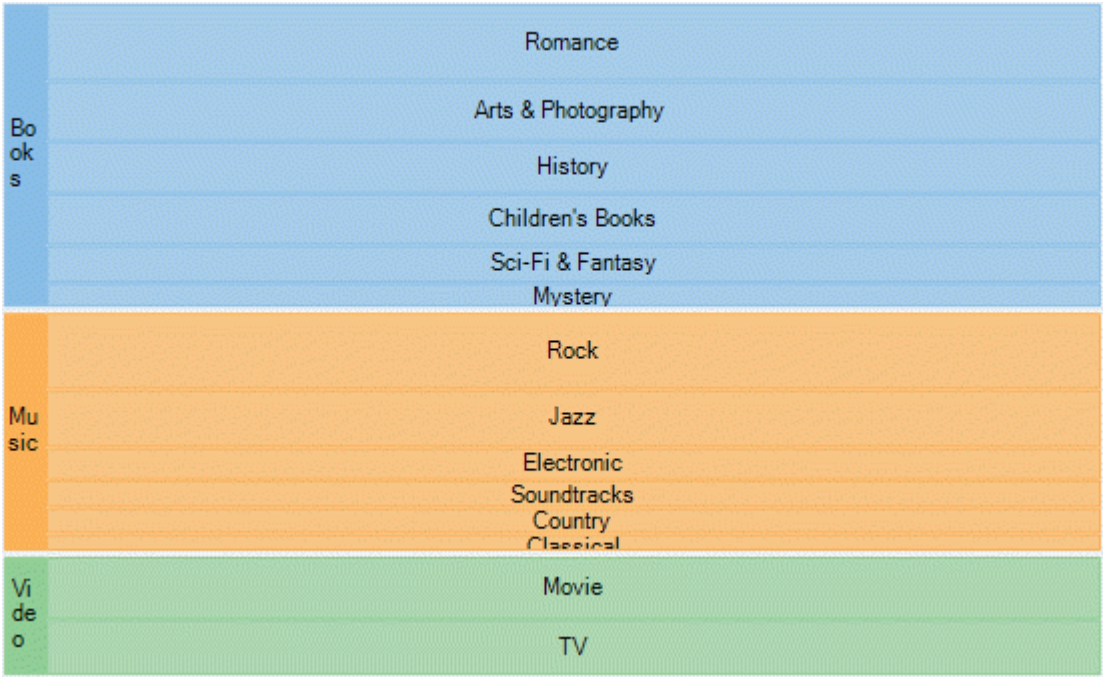


Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
XAML copyCode
<c1:C1TreeMap Binding="Value"
              BindingName="Name"
              ChartType="Squarified">
```

Horizontal

The horizontal layout stacks the tree map rectangles one over the other as rows. Here the width of the rectangles is greater than their height.

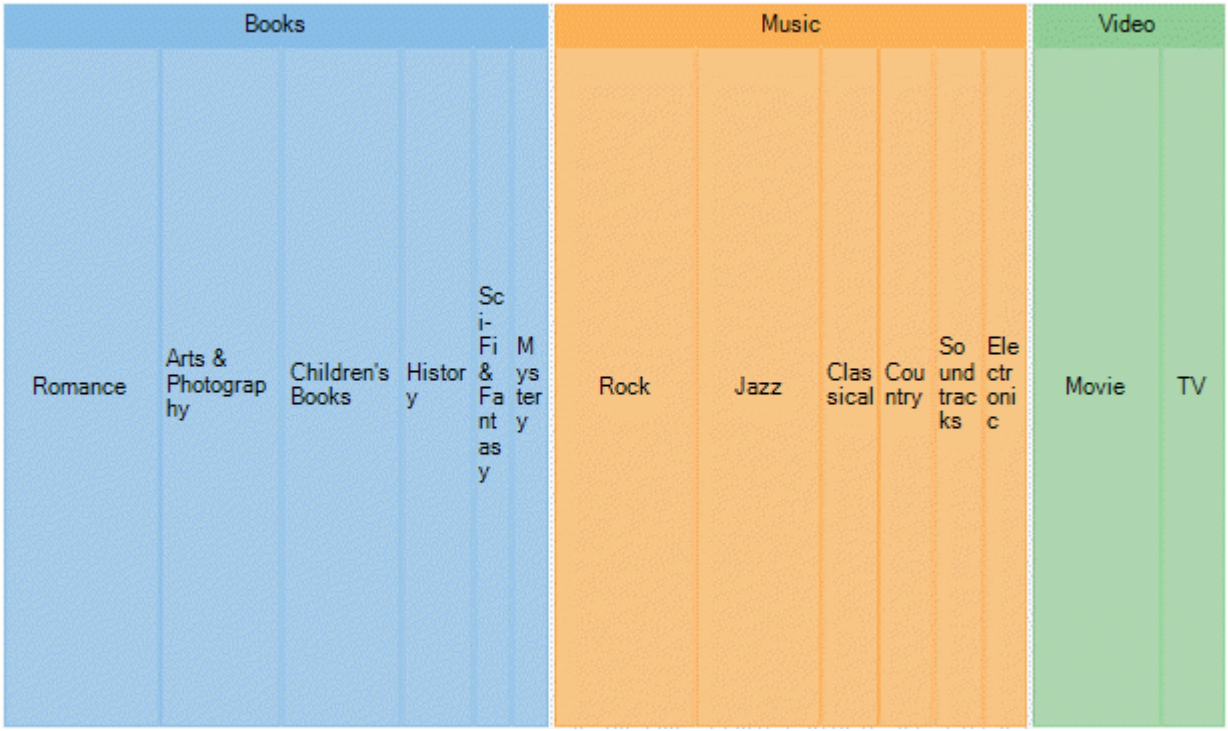



```
XAML copyCode
<c1:C1TreeMap Binding="Value"
              BindingName="Name"
              ChartType="Horizontal">
```

Vertical

The vertical layout arranges the tree map rectangles adjacent to each other as columns. Here the height of the rectangles is greater than their width.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



XAML copyCode

```
<c1:C1TreeMap Binding="Value"
              BindingName="Name"
              ChartType="Vertical">
```

Horizontal and vertical treemaps are helpful in preserving and displaying the order of information.

Data Binding

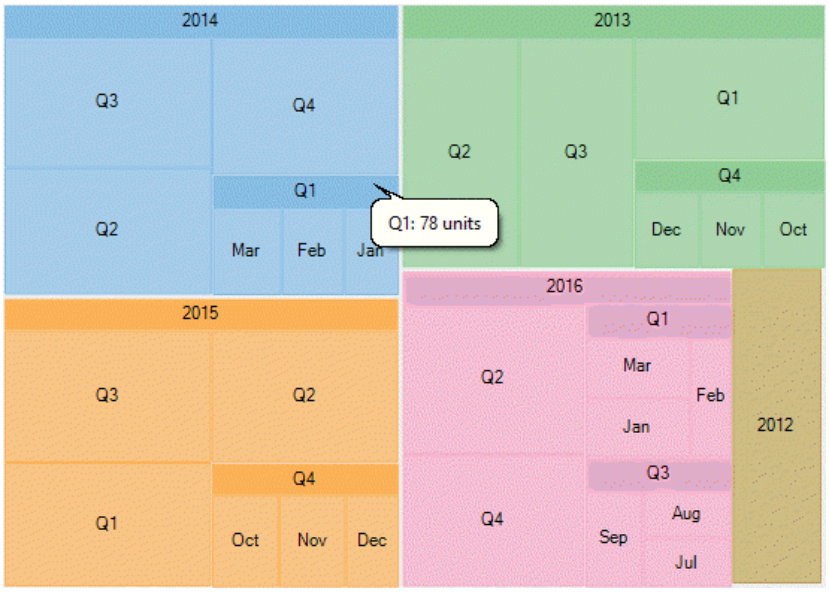
The TreeMap chart control binds to hierarchical data, to represent the elements of tree-like data as nested rectangles. Once the control binds to the data source and displays data items as rectangles, the size and color of these constituting rectangles enable analysis and comparison of data items.

FlexChartBase class exposes ItemsSource property, which takes collection of the objects, that contain data, to populate in tree map chart. The Binding and BindingName properties are instrumental in generating rectangular nodes for data items and their respective categories or groups. While **Binding** property takes string value depicting the name of the property of data item that contains numeric data value, helpful in calculating the size of rectangular nodes, **BindingName** takes string value depicting the name of data items. **ChildItemPath** property ensures that a hierarchical structure of the provided data collection is maintained, by communicating to the control about the child items within the data.

To elaborate how data is populated in a tree map chart, let's consider a use case where we try to compare yearly sales (in units sold) of a multi-brand retail store. The analysis can then further be drilled down to quarters in a year and then to months in a quarter, by using Treemap chart. Here yearly sales are represented by the top level rectangles, quarterly sales in those years represent the subsequent level, and monthly sales form the next level that is leaf nodes in tree map.

The following image illustrates sales in a retail store, in terms of units sold, through TreeMap chart control. Note that the image shows hierarchical data up to third level, that is months in respective quarters of the years.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



Back to Top

In this example, data generated in DataService class is serving as the source for tree map chart.

1. **Create a hierarchical data source**
    1. In the code view, create a DataService class to generate hierarchical data, as shown in the following code.
        - **Visual Basic**

```
Public Class DataService
    Private rnd As New Random()
    Shared _default As DataService

    Public Shared ReadOnly Property Instance() As DataService
        Get
            If _default Is Nothing Then
                _default = New DataService()
            End If

            Return _default
        End Get
    End Property

    Public Shared Function CreateHierarchicalData() As List(Of DataItem)
        Dim rnd As Random = Instance.rnd

        Dim years As New List(Of String)()
        Dim times As New List(Of List(Of String))() From {
            New List(Of String)() From {
                "Jan",
                "Feb",
                "Mar"
            },
            New List(Of String)() From {
                "Apr",
                "May",
                "June"
            },
            New List(Of String)() From {
                "Jul",
                "Aug",
                "Sep"
            },
            New List(Of String)() From {
                "Oct",
                "Nov",
                "Dec"
            }
        }

        Dim items As New List(Of DataItem)()
        Dim yearLen = Math.Max(CInt(Math.Round(Math.Abs(5 - Instance.rnd.NextDouble() * 10))), 3)
        Dim currentYear As Integer = DateTime.Now.Year
        For i As Integer = yearLen To 1 Step -1
            years.Add((currentYear - i).ToString())
        Next
        Dim quarterAdded = False
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
        years.ForEach(Function(y)
                          Dim i = years.IndexOf(y)
                          Dim addQuarter = Instance.rnd.NextDouble() > 0.5
                          If Not quarterAdded AndAlso i = years.Count - 1 Then
                              addQuarter = True
                          End If
                          Dim year = New DataItem() With {
                              .Year = y
                          }
                          If addQuarter Then
                              quarterAdded = True
                              times.ForEach(Function(q)
                                                Dim addMonth = Instance.rnd.NextDouble() > 0.5
                                                Dim idx As Integer = times.IndexOf(q)
                                                Dim quar As String
                                                quar = "Q" + (idx + 1).ToString
                                                Dim quarters = New DataItem() With {
                                                    .Year = y,
                                                    .Quarter = quar
                                                }
                                                If addMonth Then
                                                    q.ForEach(Function(m)
                                                                  quarters.Items.Add(New DataItem() With {
                                                                      .Year = y,
                                                                      .Quarter = quar,
                                                                      .Month = m,
                                                                      .Value = rnd.[Next](20, 30)
                                                                  })

                                                              End Function)
                                                Else
                                                    quarters.Value = rnd.[Next](80, 100)
                                                End If
                                                year.Items.Add(quarters)

                                            End Function)
                          Else
                              year.Value = rnd.[Next](80, 100)
                          End If
                          items.Add(year)

                      End Function)

        Return items
    End Function

End Class
```

- **C#**

```
public class DataService
{
    Random rnd = new Random();
    static DataService _default;

    public static DataService Instance
    {
        get
        {
            if (_default == null)
            {
                _default = new DataService();
            }

            return _default;
        }
    }

    public static List<DataItem> CreateHierarchicalData()
    {
        Random rnd = Instance.rnd;

        List<string> years = new List<string>();
        List<List<string>> times = new List<List<string>>()
        {
            new List<string>() { "Jan", "Feb", "Mar"},
            new List<string>() { "Apr", "May", "June"},
            new List<string>() { "Jul", "Aug", "Sep"},
            new List<string>() { "Oct", "Nov", "Dec" }
        };

        List<DataItem> items = new List<DataItem>();
        var yearLen = Math.Max((int)Math.Round(Math.Abs(5 - Instance.rnd.NextDouble() * 10)), 3);
        int currentYear = DateTime.Now.Year;
        for (int i = yearLen; i > 0; i--)
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
            {
                years.Add((currentYear - i).ToString());
            }
            var quarterAdded = false;

            years.ForEach(y =>
            {
                var i = years.IndexOf(y);
                var addQuarter = Instance.rnd.NextDouble() > 0.5;
                if (!quarterAdded && i == years.Count - 1)
                {
                    addQuarter = true;
                }
                var year = new DataItem() { Year = y };
                if (addQuarter)
                {
                    quarterAdded = true;
                    times.ForEach(q =>
                    {
                        var addMonth = Instance.rnd.NextDouble() > 0.5;
                        int idx = times.IndexOf(q);
                        var quar = "Q" + (idx + 1);
                        var quarters = new DataItem() { Year = y, Quarter = quar };
                        if (addMonth)
                        {
                            q.ForEach(m =>
                            {
                                quarters.Items.Add(new DataItem()
                                {
                                    Year = y,
                                    Quarter = quar,
                                    Month = m,
                                    Value = rnd.Next(20, 30)
                                });
                            });
                        }
                        else
                        {
                            quarters.Value = rnd.Next(80, 100);
                        }
                        year.Items.Add(quarters);
                    });
                }
                else
                {
                    year.Value = rnd.Next(80, 100);
                }
                items.Add(year);
            });

            return items;
        }

    }
```

2. Create a DataItem class to define list of objects to represent data items and their categories.
    - **Visual Basic**

```
Public Class DataItem
    Private _items As List(Of DataItem)

    Public Property Year() As String
        Get
            Return m_Year
        End Get
        Set
            m_Year = Value
        End Set
    End Property
    Private m_Year As String
    Public Property Quarter() As String
        Get
            Return m_Quarter
        End Get
        Set
            m_Quarter = Value
        End Set
    End Property
    Private m_Quarter As String
    Public Property Month() As String
        Get
            Return m_Month
        End Get
        Set
            m_Month = Value
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
            End Set
        End Property
        Private m_Month As String
        Public Property Value() As Double
            Get
                Return m_Value
            End Get
            Set
                m_Value = Value
            End Set
        End Property
        Private m_Value As Double
        Public ReadOnly Property Items() As List(Of DataItem)
            Get
                If _items Is Nothing Then
                    _items = New List(Of DataItem)()
                End If

                Return _items
            End Get
        End Property
    End Class
```

- **C#**

```
public class DataItem
{
    List<DataItem> _items;

    public string Year { get; set; }
    public string Quarter { get; set; }
    public string Month { get; set; }
    public double Value { get; set; }
    public List<DataItem> Items
    {
        get
        {
            if (_items == null)
            {
                _items = new List<DataItem>();
            }

            return _items;
        }
    }
}
```

Back to Top

2. **Bind TreeMap to the data source**

   To bind the TreeMap control to the data source use the following code.

XAML copyCode

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:WpfTreeMapCS"
    xmlns:c1="http://schemas.componentone.com/winfx/2006/xaml"
    x:Class="WpfTreeMapCS.DataBinding"
    DataContext="{Binding RelativeSource={RelativeSource Mode=Self}}"
    mc:Ignorable="d"
    Title="DataBinding" Height="300" Width="300">
<Grid>
    <Grid.DataContext>
        <local:TreeMapViewModel />
    </Grid.DataContext>
    <c1:C1TreeMap Binding="Value"
                  BindingName="Year,Quarter,Month"
                  ChildItemsPath="Items"
                  ItemsSource="{Binding HierarchicalData}"
                  MaxDepth="3">
        <c1:C1TreeMap.DataLabel>
            <c1:DataLabel Content="{}{name}"
                          Position="Center">
                <c1:DataLabel.Style>
```

Copyright © 2019 GrapeCity, Inc. All rights reserved.

```
                        <c1:ChartStyle/>
                    </c1:DataLabel.Style>
                </c1:DataLabel>
            </c1:C1TreeMap.DataLabel>
        </c1:C1TreeMap>
    </Grid>
</Window>
```

Back to Top

- **Visual Basic**

```
Public Class TreeMapViewModel
    Public ReadOnly Property HierarchicalData() As List(Of DataItem)
        Get
            Return DataService.CreateHierarchicalData()
        End Get
    End Property
End Class
```

- **C#**

```
public class TreeMapViewModel
{
    public List<DataItem> HierarchicalData
    {
        get
        {
            return DataService.CreateHierarchicalData();
        }
    }
}
```

3. **Build and run the project**
   1. Click **Build | Build Solution** to build the project.
   2. Press **F5** to run the project.

Back to Top

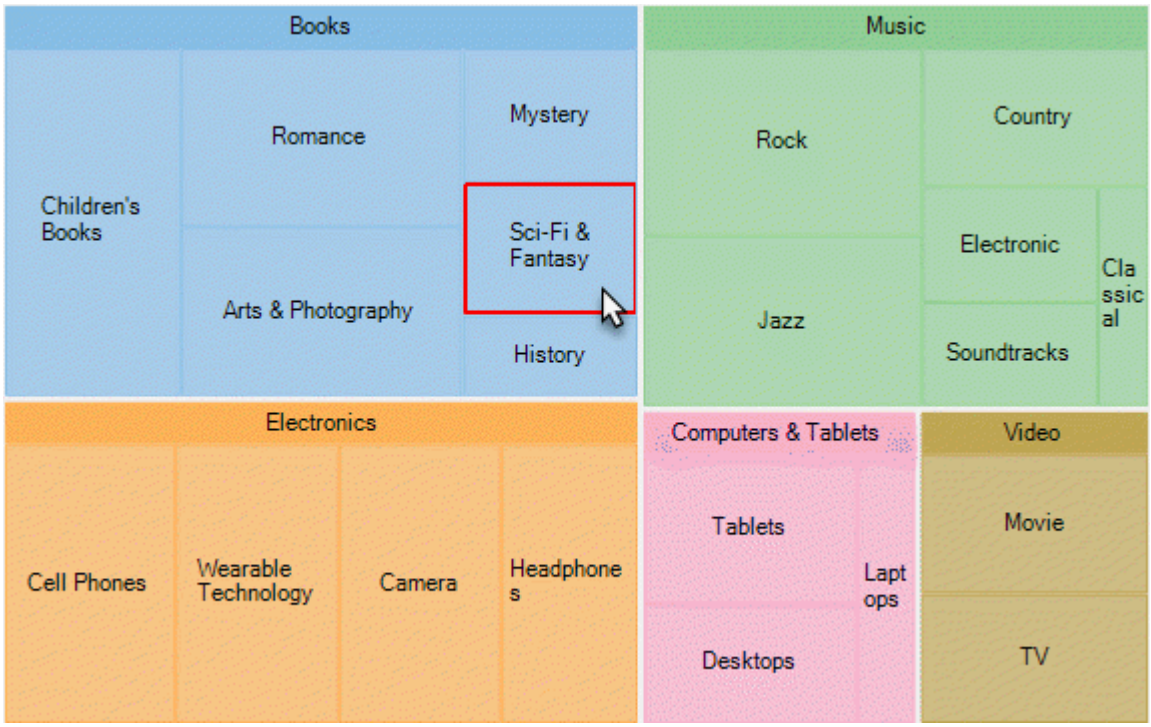
Selection

TreeMap chart lets you enable selection of its data items and groups. User can select a node and draw focus on it by simply clicking it. You need to set the SelectionMode property provided by the FlexChartBase class to either of the following values in the ChartSelectionMode enumeration:

- **None (default)**: Selection is disabled.
- **Point**: A point is selected.

The following image illustrates default selection of data point in TreeMap.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



The following code snippet shows how to set the **SelectionMode** property for a tree map chart.

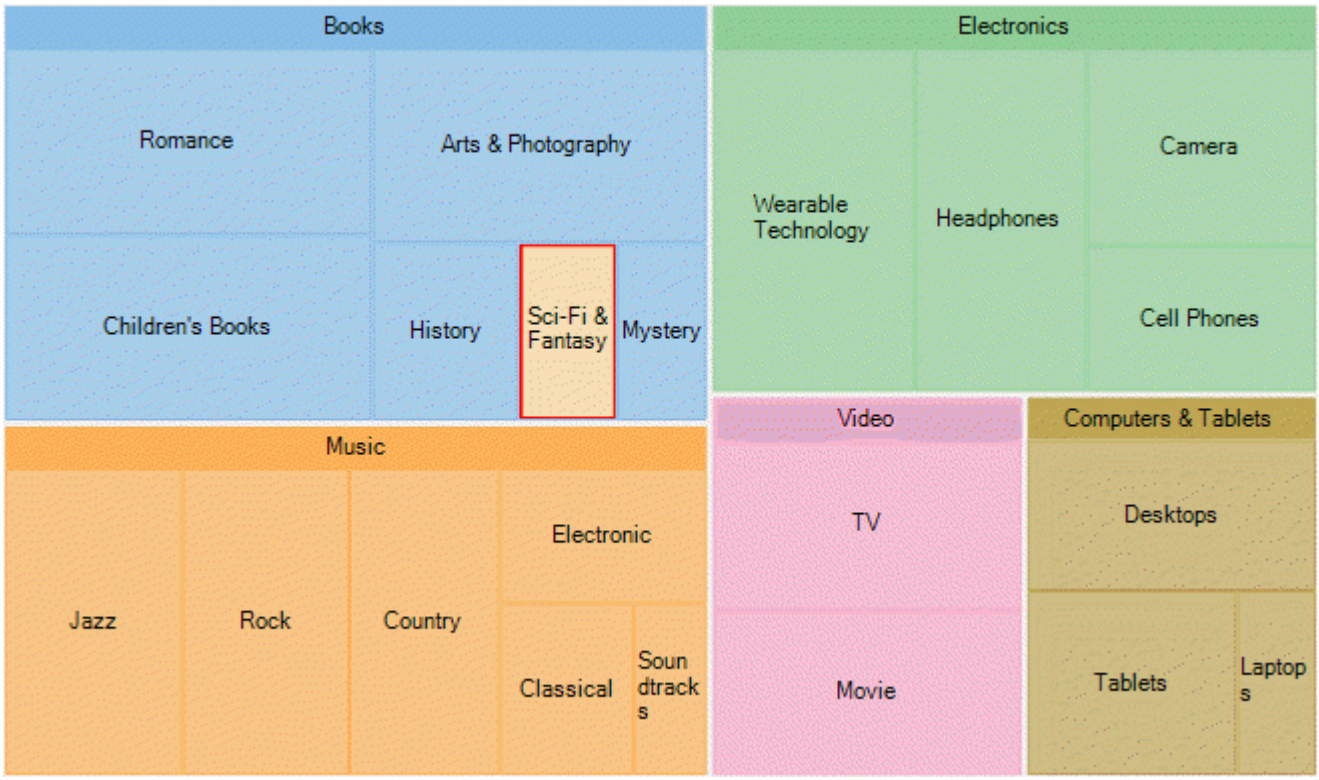
XAML copyCode

```
<c1:C1TreeMap Binding="sales"
              MaxDepth="2"
              BindingName="type"
              ChildItemsPath="items"
              ItemsSource="{Binding DataContext.Data}"
              SelectionMode="Point">
```

Customized TreeMap Selection

To customize the TreeMap selection, you can use SelectionStyle property and stylize the selected item as illustrated in the following image.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



The following code snippet demonstrates utilizing **SelectionStyle** property to change fill color of the TreeMap node that is selected.

XAML copyCode

```
<c1:C1TreeMap Binding="sales"
              MaxDepth="2"
              BindingName="type"
              ChildItemsPath="items"
              ItemsSource="{Binding DataContext.Data}"
              SelectionMode="Point">
    <c1:C1TreeMap.SelectionStyle>
        <c1:ChartStyle Fill="Wheat"/>
    </c1:C1TreeMap.SelectionStyle>
```

Additionally, you can customize the behavior of TreeMap selection by handling SelectionChanged event. Also, you can utilize SelectedIndex and SelectedItem properties, and reuse the obtained information in your application.

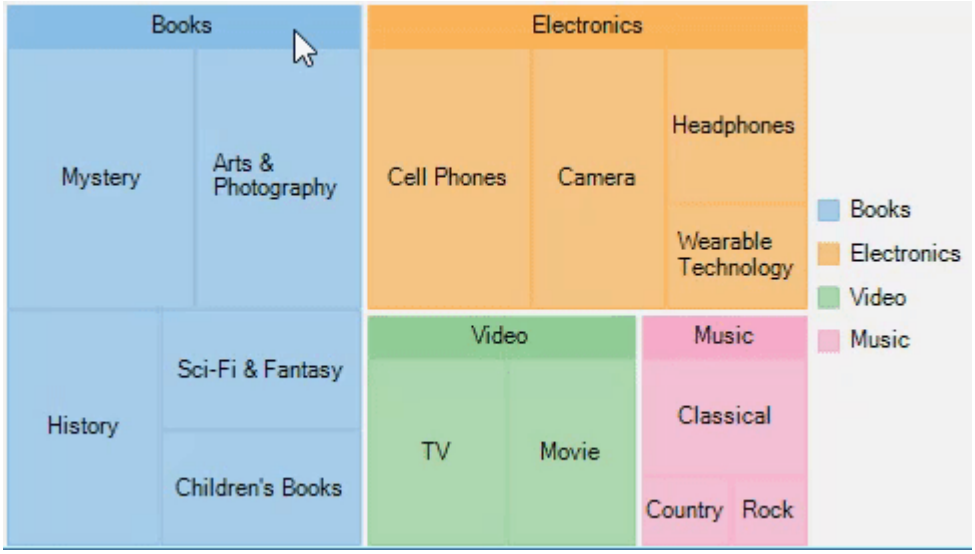
Drilldown

TreeMap allows drilling down the data items of its data further for detailed analysis. End users can access the lower levels in the data hierarchy by simply clicking the desired node. Whereas, to move back up in the hierarchy, users simply need to right-click in the plot area.

Note that, drilldown functionality in TreeMap is possible only if MaxDepth property is set to a value greater than 0. This property defines the levels of hierarchical data in the TreeMap chart.

The following gif image demonstrates drilling-down by showing data points of the clicked TreeMap node.

Copyright © 2019 GrapeCity, Inc. All rights reserved.



Note that drill down feature of TreeMap works only when selection of TreeMap nodes is disabled, that is, SelectionMode property is set to **None**. For more information on selection, see Selection in TreeMap.

Copyright © 2019 GrapeCity, Inc. All rights reserved.