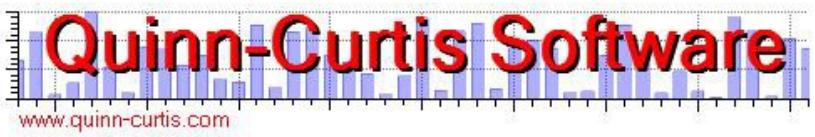
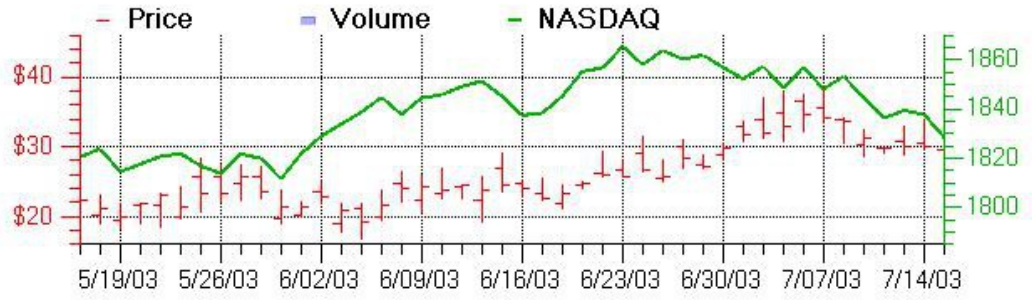
QCChart2D Charting Tools

for Windows Apps





"On inspecting any one of these Charts attentively, a sufficiently distinct impression will be made, to remain unimparied for a considerable time, and the idea which does remain will be simple and complete, at once including the duration and the amount."

William Playfair, the father of statistical graphics, in 1786

Contact Information

Company Web Site: **http://**www.quinn-curtis.com

Electronic mail

General Information: info@quinn-curtis.com
Sales: sales@quinn-curtis.com

Technical Support Forum

http://www.quinn-curtis.com/ForumFrame.htm

Revision Date 4/15/2016 Rev. 2.5

Quinn-Curtis, Inc. Tools for *Windows Apps* END-USER LICENSE AGREEMENT

IMPORTANT-READ CAREFULLY: This Software End-User License Agreement ("EULA") is a legal agreement between you (either an individual or a single entity) and Quinn-Curtis, Inc. for the Quinn-Curtis, Inc. SOFTWARE identified above, which includes all Quinn-Curtis, Inc. ***.Net*** software (on any media) and related documentation (on any media). By installing, copying, or otherwise using the SOFTWARE, you agree to be bound by the terms of this EULA. If you do not agree to the terms of this EULA, do not install or use the SOFTWARE. If the SOFTWARE was mailed to you, return the media envelope, UNOPENED, along with the rest of the package to the location where you obtained it within 30 days from purchase.

1. The SOFTWARE is licensed, not sold.

2. GRANT OF LICENSE.

(A) Developer License. After you have purchased the license for SOFTWARE, and have received the file containing the licensed copy, you are licensed to copy the SOFTWARE only into the memory of the number of computers corresponding to the number of licenses purchased. The primary user of the computer on which each licensed copy of the SOFTWARE is installed may make a second copy for his or her exclusive use on a portable computer. Under no other circumstances may the SOFTWARE be operated at the same time on more than the number of computers for which you have paid a separate license fee. You may not duplicate the SOFTWARE in whole or in part, except that you may make one copy of the SOFTWARE for backup or archival purposes. You may terminate this license at any time by destroying the original and all copies of the SOFTWARE in whatever form.

(B) 30-Day Trial License. You may download and use the SOFTWARE without charge on an evaluation basis for thirty (30) days from the day that you DOWNLOAD the trial version of the SOFTWARE. The termination date of the trial SOFTWARE is embedded in the downloaded SOFTWARE and cannot be changed. You must pay the license fee for a Developer License of the SOFTWARE to continue to use the SOFTWARE after the thirty (30) days. If you continue to use the SOFTWARE after the thirty (30) days without paying the license fee you will be using the SOFTWARE on an unlicensed basis.

Redistribution of 30-Day Trial Copy. Bear in mind that the 30-Day Trial version of the SOFTWARE becomes invalid 30-days after downloaded from our web site, or one of our sponsor's web sites. If you wish to redistribute the 30-day trial version of the SOFTWARE you should arrange to have it redistributed directly from our web site If you are using SOFTWARE on an evaluation basis you may make copies of the evaluation SOFTWARE as you wish; give exact copies of the original evaluation SOFTWARE to anyone; and distribute the evaluation SOFTWARE in its unmodified form via electronic means (Internet, BBS's, Shareware distribution libraries, CD-ROMs, etc.). You may not charge any fee for the copy or use of the evaluation SOFTWARE itself. You must not represent in any way that you are selling the SOFTWARE itself. You must distribute a copy of this EULA with any copy of the SOFTWARE and anyone to whom you distribute the SOFTWARE is subject to this EULA.

(C) Redistributable License. The standard Developer License permits the programmer to deploy and/or distribute applications that use the Quinn-Curtis SOFTWARE, royalty free. We cannot allow developers to use this SOFTWARE to create a graphics toolkit (a library or any type of graphics component that will be used in combination with a program development environment) for resale to other developers.

If you utilize the SOFTWARE in an application program, or in a web site deployment, should we ask, you must supply Quinn-Curtis, Inc. with the name of the application program and/or the URL where the SOFTWARE is installed and being used.

3. RESTRICTIONS. You may not reverse engineer, de-compile, or disassemble the SOFTWARE, except and only to the extent that such activity is expressly permitted by applicable law notwithstanding this limitation. You may not rent, lease, or lend the SOFTWARE. You may not use the SOFTWARE to perform any illegal purpose.

4. SUPPORT SERVICES. Quinn-Curtis, Inc. may provide you with support services related to the SOFTWARE. Use of Support Services is governed by the Quinn-Curtis, Inc. polices and programs described in the user manual, in

online documentation, and/or other Quinn-Curtis, Inc.-provided materials, as they may be modified from time to time. Any supplemental SOFTWARE code provided to you as part of the Support Services shall be considered part of the SOFTWARE and subject to the terms and conditions of this EULA. With respect to technical information you provide to Quinn-Curtis, Inc. as part of the Support Services, Quinn-Curtis, Inc. may use such information for its business purposes, including for product support and development. Quinn-Curtis, Inc. will not utilize such technical information in a form that personally identifies you.

5. TERMINATION. Without prejudice to any other rights, Quinn-Curtis, Inc. may terminate this EULA if you fail to comply with the terms and conditions of this EULA. In such event, you must destroy all copies of the SOFTWARE.

6. COPYRIGHT. The SOFTWARE is protected by United States copyright law and international treaty provisions. You acknowledge that no title to the intellectual property in the SOFTWARE is transferred to you. You further acknowledge that title and full ownership rights to the SOFTWARE will remain the exclusive property of Quinn-Curtis, Inc. and you will not acquire any rights to the SOFTWARE except as expressly set forth in this license. You agree that any copies of the SOFTWARE will contain the same proprietary notices which appear on and in the SOFTWARE.

7. EXPORT RESTRICTIONS. You agree that you will not export or re-export the SOFTWARE to any country, person, entity, or end user subject to U.S.A. export restrictions. Restricted countries currently include, but are not necessarily limited to Cuba, Iran, Iraq, Libya, North Korea, Sudan, and Syria. You warrant and represent that neither the U.S.A. Bureau of Export Administration nor any other federal agency has suspended, revoked or denied your export privileges.

8. NO WARRANTIES. Quinn-Curtis, Inc. expressly disclaims any warranty for the SOFTWARE. THE SOFTWARE AND ANY RELATED DOCUMENTATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OR MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. THE ENTIRE RISK ARISING OUT OF USE OR PERFORMANCE OF THE SOFTWARE REMAINS WITH YOU.

9. LIMITATION OF LIABILITY. IN NO EVENT SHALL QUINN-CURTIS, INC. OR ITS SUPPLIERS BE LIABLE TO YOU FOR ANY CONSEQUENTIAL, SPECIAL, INCIDENTAL, OR INDIRECT DAMAGES OF ANY KIND ARISING OUT OF THE DELIVERY, PERFORMANCE, OR USE OF THE SUCH DAMAGES. IN ANY EVENT, QUINN-CURTIS'S LIABILITY FOR ANY CLAIM, WHETHER IN CONTRACT, TORT, OR ANY OTHER THEORY OF LIABILITY WILL NOT EXCEED THE GREATER OF U.S. $1.00 OR LICENSE FEE PAID BY YOU.

10. U.S. GOVERNMENT RESTRICTED RIGHTS. The SOFTWARE is provided with RESTRICTED RIGHTS. Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of The Rights in Technical Data and Computer SOFTWARE clause of DFARS 252.227-7013 or subparagraphs (c)(i) and (2) of the Commercial Computer SOFTWARE- Restricted Rights at 48 CFR 52.227-19, as applicable. Manufacturer is: Quinn-Curtis, Inc., 18 Hearthstone Dr., Medfield MA 02052 USA.

11. MISCELLANEOUS. If you acquired the SOFTWARE in the United States, this EULA is governed by the laws of the state of Massachusetts. If you acquired the SOFTWARE outside of the United States, then local laws may apply.

Should you have any questions concerning this EULA, or if you desire to contact Quinn-Curtis, Inc. for any reason, please contact Quinn-Curtis, Inc. by mail at: Quinn-Curtis, Inc., 18 Hearthstone Dr., Medfield MA 02052 USA, or by telephone at: (508)359-6639, or by electronic mail at: support@Quinn-Curtis.com.

QCChart2D for Windows Apps Table of Contents

The QCChart2D for Windows Apps Charting Library

1. Introduction

A point of clarification. What is today called *Windows Apps* has also be referred to officially by Microsoft over the past couple of year under many other names: Windows 8 Apps, Metro Apps, Windows Store Apps, UWP, Universal Windows Apps, Windows Universal Platform Apps. As of March 2015, Microsoft rebranded this collection of products as Windows Apps. We will refer to this class of application programs as Windows Apps, and using the abbreviation UWP (Universal Windows Platform), which still seems to be in common use.

A Window App is designed to run on a single unifying Windows platform, known as the Windows Runtime platform, which was introduced with Windows 8. The design goal is that the Windows App user can move a program across the entire family of Microsoft products which can run one of the Windows 10 operating systems. Some of the characteristics highlighted by Microsoft on their web site (https://msdn.microsoft.com/en-us/library/windows/apps/dn726767.aspx) are :

- " You target device families, not an OS.

  A device family identifies the APIs, system characteristics, and behaviors that you can expect across devices within the device family. It also determines the set of devices on which your app can be installed from the store.

- Apps are packaged and distributed using the .AppX packaging format.

  All UWP apps are distributed as an AppX package. This provides a trustworthy installation mechanism and ensures that your apps can be deployed and updated seamlessly.

- There's one store for all devices.

  After you register as an app developer, you can submit your app to the store and make it available on all device families, or only those you choose. You submit and manage all your apps for Windows devices in one place.

- There's a common API surface across device families.

  The Windows (UWP) core APIs are the same for all Windows device families. If your app uses only the core APIs, it will run on any Windows 10 device.

- Extension SDKs make your app light up on specialized devices.

Extension SDKs add specialized APIs for each device family. If your app is intended for a particular device family, you can make it light up by using these APIs. You can still have one app package that runs on all devices by checking what device family your app is running on before calling an extension API.

- Adaptive Controls and input

  UI elements use *effective pixels* (see Responsive design 101 for UWP apps), so they automatically adapt themselves based on the number of screen pixels available on the device. And they work well with multiple types of input such as keyboard, mouse, touch, pen, and Xbox One controllers. If you need to further tailor your UI to a specific screen size or device, new layout panels and tooling help you adapt your UI to the devices your app may run on. "

The **QCChart2D for Windows Apps** software represents an adaptation of the **QCChart2D** library to the Windows Apps user interface framework. We have removed 100% of the GDI+ based graphics found in the .**Net System.Drawing**, **System.Drawing.Drawing2D** and **System.Windows.Forms** names-spaces, and replaced them with Universal Windows Platform (USP) equivalents. We have redesigned the chart rendering scheme to work with the Windows Apps retained graphics framework. But, we have maintained the simple to use, flexible programming style found in our QCChart2D for .Net software. In general, you place one or more of our ChartView objects as visual elements in the XAML window of your application. The chart itself is customized in the behind code of the XAML form.

Tutorials

Tutorials that describe how to get started with the QCChart2D for Windows Apps charting software are found in Chapter 24 (*Using QCChart2D for Windows Apps to Create Windows Applications*).

Customer Support

Use our forums at http://www.quinn-curtis.com/ForumFrame.htm for customer support. Please, do not post questions on the forum unless you are familiar with this manual and have run the examples programs provided. We try to answer most questions by referring to the manual, or to existing example programs. We will always attempt to answer any question that you may post, but be prepared that we may ask you to create, and send to us, a simple example program. The program should reproduce the problem with no, or minimal interaction, from the user. You should strip out of any code not directly associated with reproducing the problem. You can use either your own example or a modified version of one of our own examples.

Windows Apps Background

Initially, the primary graphics, text rendering, and user interface framework for programmers using the .Net languages was GDI+, an evolutionary adaptation of the older Windows GDI programming model in place since Windows 1.0. GDI+ under .Net is extremely successful, and there are no indications Microsoft plans to end support for it in subsequent releases of Visual Studio. But, Microsoft has long had an alternative graphics framework for game programmers, DirectX, which programmers found a better fit for the complex graphics required in game programming. One of the strong selling points of DirectX is that the Windows operating system can offload time-intensive graphics calculations to specialized GPU (Graphics Processing Unit) chips, found in high- and medium-end

computers. Starting early in the last decade, Microsoft wrote an alternative graphics rendering and user interface framework around DirectX. This framework is known as the Windows Presentation Foundation framework, or WPF for short. At the same time, they also created a similar framework for use in Web Applications and that was called Silverlight. Unfortunately, Microsoft has discontinued development on Silverlight, a bad decision in our opinion. Instead, for web programming they are focusing on HTML5 and Javascript. Perhaps Microsoft feels it can improve the development environment (Visual Studio) for HTML5 and Javascript more than other companies can. WPF continues, but many feel that it too is being phased out – in favor of the Universal Windows App framework (UWP), which is very similar to WPF and Silverlight, since all three use a XAML-centered UI framework. From what we can tell, DirectX is not part of the regular .Net/C#/Windows App framework. Instead, the UniversalWindows App support libraries integrate their own graphics framework from what Microsoft calls the Windows Runtime.

Graphics features used in the **QCChart2D for Windows Apps** library include the following:

- Resolution independence. An emphasis on vector graphics means that programs can be more easily designed to be independent of the resolution of the output device.
- Arbitrary line thickness and line styles for all lines.
- Gradients, fill patterns and color transparency for solid objects.
- Generalized geometry support used to create arbitrary shapes
- Printer and image output support
- Improved font support for a large number of fonts, using a variety of font styles, size and rotation attributes.
- Imaging support for a large number of image formats
- Advanced matrix support for handling 2D transformations.

In addition to graphics, Windows Apps utilizes a declarative user interface definition framework known as XAML (for Extensible Application Markup Language). This framework combines customizable rendering of user interface components with a two tier programming model, analogous to the the way you program ASP.Net web pages using a combination of HTML in the design mode, and C# or VB in the behind code section of the page. In Windows Apps applications, the user interface layout is defined as an XAML page using a text/tag format based on XML, and user interface events are processed in a C# behind code page.

In a Windows App program which uses QCChart2D, the QCChart2D class ChartView is referenced as a visual element in a windows the XAML file, as below.

```
<Page
    x:Class="WAChartApplication1.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:WAChartApplication1"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
   xmlns:my="using:com.quinncurtis.chart2dwa"
    mc:Ignorable="d">
    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
        <my:ChartView x:Name="simplelineandscatterplot1" HorizontalAlignment="Stretch"
            Margin="10,10,10,10" VerticalAlignment="Stretch"/>
    </Grid>
</Page>
```

In this example, the **ChartView** is placed as the only visual element of the standard Windows Apps layout panel, **Grid**. Since the **ChartView** object is given a "Name", it can be accessed in the behind code of the window using the variable name *simplelineandscatterplot1*. In general, all of the chart definition and initialization takes place in the the behind code, using either C# or VB.

The positioning and size are controlled by the HorizontalAlignment and VerticalAlignment properties, which are set to Stretch, which means that the chart will fill the area of the parent. This allows a large degree of device independence, and works well with grid cells, which can be assigned a relative size to the parent grid. In the case of other containers you may have to assign a specific size.

The chart definition/initialization can take place entirely in the behind code of the windows XAML file, or it be done in a separate class created for that purpose. Almost all of our examples use a separate class to initialize the **ChartView** object. This keeps the code more modular and the Window file much easier to read. In the behind code example below, extracted from our WAChartApplication1 example, the **ChartView** object (variable name chartView1 in the example) is initialized using the class **SimpleLineAndScatterPlot**. The **SimpleLineAndScatterPlot** class adds the coordinate systems, axes, plots and text to the chartView1 object, turning it into requisite chart.

```
namespace WAChartApplication1
{
    /// <summary>
    /// An empty page that can be used on its own or navigated to within a Frame.
    /// </summary>
    public sealed partial class MainPage : Page
    {

        SimpleLineAndScatterPlot slsp = null;

        public MainPage()
        {
            this.InitializeComponent();
            InitChartView();
        }

        public void InitChartView()
        {


            slsp = new SimpleLineAndScatterPlot(simplelineandscatterplot1);

        }
    }
}
```

The instance of the *simplelineandscatterplot1* variable comes from the associated XAML file, where it is defined using the Name property of the ChartView object placed in the Grid of the XAML layout.

Directory Structure of QCChart2D for Windows Apps

The QCChart2D for Windows Apps class library uses the following directory structure:

Drive:

Quinn-Curtis\ - Root directory

DotNet\ - Quinn-Curtis .Net based products directory

Docs\ - Quinn-Curtis .Net related documentation directory

Lib\ - Quinn-Curtis .Net related compiled libraries and components directory

QCChart2D\ - Language specific code directory

Visual CSharp\ - C# specific directory

QCChart2DWAF3\ - contains the source code to the QCChart2DWA.dll library (installed only if the source code has been purchased)

Examples wa\ - C# examples directory

Bargraphs\ - Horizontal, vertical, stacked, group, histogram, floating

CalendarData\ - Line plots, bar plots and log plots that use time/date data. Also reading time/date data from a file.

ChartAxes\ - Linear, logarithmic, time/date, custom hours time/date and polar axes. Also axes labels.

ChartEventExamples\ - A variety of examples using ChartEvent objects as the source data

ContourPlots\ - Line and filled contour plots.

CustomDataToolTips\ - Creating custom data tooltips for an OHLC plot, multiple stacked graphs and a pie chart.

DataCursorsAndMarkers\ - Using data cursors and markers

DynamicCharts\ - Scrolling lines, bars, scatter plots, data logging, instrument simulation, chart animation.

EditChartExample\ - Dialog box for chart example

FinancialExamples\ - OHLC plots, candlestick plots, financial log plots, option chart, technical analysis chart.

FormControlExamples\ - Adding check boxes, scrollbars to charts.

ImageCharts\ - Using images as chart data elements, chart backgrounds and annotations.

LinePlotSalesVolume\ - Simple line plot example with printing and save image menu.

LogPlots\ -Logarithmic plots for financial charts and engineering charts.

MainPageChartExample\ - A simple chart example

MiscCharts\ - A line gap chart.

MouseListeners\ - Data tooltips, data cursors, moving data points, moving chart objects.

MultiLinePlots\ - Group multi-line plots, stacked line plots, multiple single line graphs.

MultipleAxes\ - Multiple axes graphs

NewDemosRev2 – New examples for Revision 2.0 features.

PieCharts\ - Simple pie charts and pie charts combined with line and bar plots.

PolarCharts\ - Polar line, line file and scatter plots. Also includes an Antenna plot example.

PrintChartExample\ - An example which demonstrates how to print a chart.

ResizeExamples\ - Fixed size frame, resizeable frame with fixed sized objects, resizeable frame with resizeable objects, scrollable panel as a view into a much larger fixed size frame.

ScatterPlots\ - Simple scatter, line and line-marker plots, scatter plots with variable size and color symbols, cell plots, arrow plots. Also, labeling the data point values or a line marker plot.

SimpleLinePlots\ - Simple line plots with linear and time/date coordinate systems. Also filled and step lines.

WAChartApplication1\ - A simple UWP application with a single chart.

UserControlChartExample1\ - A simple UWP application with a single chart.

ZoomExamples\ - Zooming simple linear axes, super zooming of multiple axes, zooming of time/data based data.

Visual Basic\ - VB specific code

Examples wa\ - VB examples

WAChartApplication1\ - A simple UWP application with a single chart.

Visual Studio Community 2015 (It's FREE)

Because Windows App development has been in such flux, with the Visual Studio Templates for creating Windows Apps constantly changing names: Metro Apps, Store Apps, Universal Apps, we

have chosen to use Visual Studio Community 2015 as the lowest common denominator we will support. Since Visual Studio Community 2015 is a **free** download from Microsoft, nothing prevents you from downloading it in order to used the software.

(*** Critical Note ***) Running the Example Programs

The example programs for **QCChart2D Tools for Windows Apps** software are supplied in complete source. In order to save space, they have not been pre-compiled which means that many of the intermediate object files needed to view the main form are not present. This means that **ChartView** derived control will not be visible on the main Form if you attempt to view the main form before the project has been compiled. The default state for all of the example projects should be the Start Page. Before you do view any other file or form, do a build of the project. This will cause the intermediate files to be built. If you attempt to view the main Form before building the project, Visual Studio sometimes decides that the **ChartView** control placed on the main form does not exist and deletes it from the project.

There are two versions of the QCChart2D for Windows Apps class library: the 30-day trial versions, and the developer version. Each version has different characteristics that are summarized below:

30-Day Trial Version

The trial version of QCChart2D for Windows Apps is downloaded as a zip file named Trial_QCChart2DWAR20x.zip. The 30-day trial version stops working 30 days after the initial download. The trial version includes a version message in the upper left corner of the graph window that cannot be removed.

Developer Version

The developer version of QCChart2D for Windows Apps is downloaded as a zip file, name something similar to WACHTDEV1R2x0x353x1.zip. The developer version does not time out and you can use it to create application programs that you can distribute royalty free. You can download free updates for a period of 2-years. When you placed your order, you were e-mailed download link(s) that will download the software. Those download links will remain active for at least 2 years and should be used to download current versions of the software. After 2 years you may have to purchase an upgrade to continue to download current versions of the software.

Chapter Summary

The remaining chapters of this book discuss the **QCChart2D for Windows Apps** interactive charting package designed to run on any hardware that has .Net or higher interpreter, available for it.

Chapter 2 presents the overall class architecture of the **QCChart2D for Windows Apps** and summarizes all of the classes found in the software.

Chapter 3 describes the dataset classes that hold chart data.

Chapter 4 describes the various classes that implement the Cartesian, time and polar coordinate systems supported by the software.

Chapter 5 describes the **ChartView** container class that manages the chart objects**.**

Chapter 6 describes the color, gradient and background classes.

Chapters 7, 8 and 9 describe the classes that create chart axes, axis labels and axis grids.

Chapter 10 describes the classes used to display simple xy data (one y-value for each x-value) as line plots, bar plots, scatter plots., line-marker plots and simple versa plots.

Chapter 11 describes the classes used to display group data (one or more y-values for each x-value) as line plots, group bar plots, stacked bar plots, scatter plots, open-high-low-close plots, candlestick plots, box and whisker plots, floating stacked bar plots, and group versa plots.

Chapter 12 describes plotting surface data using line and filled contours.

Chapter 13, 14, 15 and 16 describe classes that add interactive elements to a chart. Chapter 13 describes data marker and data cursor classes used to mark and highlight data points using the mouse. Chapter 14 describes classes that can move chart objects, individual data points and the coordinate system. Chapter 15 adds a "zooming" class where mouse events define a new scaling range for a chart, redrawing the chart axes and data automatically. Chapter 16 describes a generalized data tool-tip class that can display the x and/or y data values for a data point, or custom text associated with the data point.

Chapter 17 describes classes for the display of pie and ring charts.

Chapter 18 describes classes for the display of data in a polar, and antenna, chart format

Chapter 19 describes classes for the display chart legends, used to create visual aids for the interpretation of different elements making up the chart.

Chapter 20 describes generalized classes for displaying formatted text in a chart.

Chapter 21 describes how the DatasetViewer class is used to display simple and group datasets in a table format.

Chapter 22 describes how to use a generalized shape class for the display of arbitrary lines, shapes, images and arrows.

Chapter 23 describes chart printing and the creation of image files.

Chapter 24 is a tutorial that describes how to use QCChart2D to create Windows applications using Visual Studio, Visual C# and Visual Basic.

Chapter 25 is a tutorial that describes how to use QCChart2D to create Windows Apps web applications using Visual Studio .Net, Visual C# and Visual Basic

Chapter 26 is a collection of Frequently Asked Questions about **QCChart2D for Windows Apps**.

2. Class Architecture of the QCChart2D for Windows Apps Class Library

Major Design Considerations

This chapter presents an overview of the **QCChart2D for Windows Apps** class architecture. Some of the major design considerations are:

- It is based on the .Net Windows App Runtime Graphics API model.
- New charting objects can be added to the library without modifying the source of the base classes.
- There are no limits regarding the number of data points in a plot, the number of plots in graph, the number of axes in a graph, the number of coordinate systems in a graph.
- There are no limits regarding the number of legends, arbitrary text annotations, bitmap images, geometric shapes, titles, data markers, cursors and grids in a graph.
- Users can interact with charts using classes using routed event handling.

The chapter also summarizes the classes in the **QCChart2D for Windows Apps** library.

There are five primary features of the overall architecture of the **QCChart2D for Windows Apps** classes. These features address major shortcomings in existing charting software for use with both .Net and other computer languages.

- First, **QCChart2D for Windows Apps** uses the standard .Net Windows App Runtime graphics architecture. Charts are placed in a **ChartView** window that derives from the **Windows.UI.Xaml.Controls.UserControl** class. Position one or more **ChartView** objects in container windows using the standard container layout managers. Mix charts with other components in the same container. Charts use the standard routed event-processing model for handling mouse and keyboard events.
- Second, the library is extensible. Hundreds of different vertical markets use computer charting. The charts used in each market have a unique look and feel. A well-designed object oriented charting package allows the programmer to extend the software without modifying the source of the underlying classes. Instead, the programmer extends the software by deriving a new class from an existing base class. The new, derived class localizes custom source code and the source of the underlying

classes remains unchanged. In the **QCChart2D for Windows Apps** classes a user can subclass an existing class and create new, custom charting objects. Examples of custom charting objects are specialized plots that extend the **SimplePlot**, or **GroupPlot** classes to include new plot types for applications such as stock market technical analysis, statistical process control, and medical instrumentation, to name a few.

- Third, the library has no limits regarding the number of data points in a plot, the number of plots in graph, the number of axes in a graph, and the number of coordinate systems in a graph. A major weakness in many commercial graphics packages is that they have hard coded limits that restrict the number of data points, axes, or coordinate systems. A simple business bar chart may only contain 3 or 4 data points, depicting a sales forecast. An audio mixer application may require 32 million plotted points, represented by 32 traces of 1 million points each, each trace representing 20 seconds of audio sampled at 50 kHz.

  The most effective way to compare data is to overlay it in the same graph. Often the data series have different dynamic ranges. If the data series are plotting using the same scale, it is difficult do see the correlation, or lack of, in the data. A better solution is to create a unique scale for each data series, with associated axes, and plot each data series with respect to its own scale. All of the plots will overlay the same area of the graph. Many advanced charting packages do not support more than one coordinate system per graph, while most others have some fixed limit such as 2, 4 or 8 scales per graph. The number of coordinate systems for a graph can be as large as the number of data series plotted in the graph. Charts can have hundreds of data series at once; therefore, a flexible charting package needs to allow for an equal number of simultaneous coordinate systems.

- Fourth, a well-constructed chart often displays more than just data. Other common chart objects include legends, arbitrary text annotations, bitmap images, geometric shapes, titles, data markers, cursors and grids. A chart can contain zero or more of these objects. It may contain 100 of one type, 5 of another type, and 1 of a third. **QCChart2D for Windows Apps** contains no limits restricting the number of instances of a given chart object in a graph, and no limit on the total number of chart objects in a graph.

- Fifth, an end user needs to interact with the graph using the mouse and/or keyboard. The **QCChart2D for Windows Apps** architecture includes classes that implement the UWP routed event model. A user can use the mouse to select data points, text annotations, axes, image objects, and other shapes, and position them in the graph. Create data markers and move them around the chart under mouse or program control. Automatically rescale one or more chart axes using mouse controlled zooming.

QCChart2D for Windows Apps Class Summary

The following categories of classes realize these design considerations.

Chart view class — The chart view class is a **Windows.UI.Xaml.Controls.UserControl** subclass that manages the graph objects placed in the graph

Data classes	There are data classes for simple xy and group data types. There are also data classes that handle **System.DateTime** date/time data and contour data.
Scale transform classes	The scale transform classes handle the conversion of physical coordinate values to working coordinate values for a single dimension.
Coordinate transform classes	The coordinate transform classes handle the conversion of physical coordinate values to working coordinate values for a parametric (2D) coordinate system.
Attribute class	The attribute class encapsulates the most common attributes (line color, fill color, line style, line thickness, etc.) for a chart object.
Auto-Scale classes	The coordinate transform classes use the auto-scale classes to establish the minimum and maximum values used to scale a 2D coordinate system. The axis classes also use the auto-scale classes to establish proper tick mark spacing values.
Charting object classes	The chart object classes includes all objects placeable in a chart. That includes axes, axes labels, plot objects (line plots, bar graphs, scatter plots, etc.), grids, titles, backgrounds, images and arbitrary shapes.
Mouse interaction classes	These classes permit the user to create and move data cursors, move plot objects, display tooltips and select data points in all types of graphs.
File and printer rendering	These classes render the chart image to a printer, to a variety of file formats including JPEG, and BMP, or to a UWP **WriteableBitmap** object.
Miscellaneous utility classes	Other classes use these for data storage, file I/O, and data processing.

A summary of each category appears in the following section.

Chart Window Classes

Windows.UI.Xaml.Controls.UserControl
    ChartView

The starting point of a chart is the **ChartView** class. The **ChartView** class derives from the Windows Runtime **Windows.UI.Xaml.Controls.UserControl** class. The **ChartView** class manages a collection of chart objects in a chart and automatically updates the chart objects whenever the control needs to redraw itself. Since the **ChartView** class is a subclass of the **UserControl** class, it can act as a container for other components, such as buttons and checkboxes.

Data Classes

ChartDataset
- **SimpleDataset**
  - **TimeSimpleDataset**
  - **ElapsedTimeSimpleDataset**
  - **ContourDataset**
  - **EventSimpleDataset**
- **GroupDataset**
  - **TimeGroupDataset**
  - **ElapsedTimeGroupDataset**
  - **EventGroupDataset**

The dataset classes organize the numeric data associated with a plotting object. There are two major types of data supported by the **ChartDataset** class. The first is simple xy data, where for every x-value there is one y-value. The second data type is group data, where every x-value can have one or more y-values.

ChartDataset	The abstract base class for the other dataset classes. It contains data common to all of the dataset classes, such as the x-value array, the number of x-values, the dataset name and the dataset type.
SimpleDataset	Represents simple xy data, where for every x-value there is one y-value.
TimeSimpleDataset	A subclass of **SimpleDataset**, it is initialized using **ChartCalendar** dates (a wrapper around the System.DateTime value class) in place of the x- or y-values.
ElapsedTimeSimpleDataset	A subclass of **SimpleDataset**, it is initialized with **TimeSpan** objects, or milliseconds, in place of the x- or y-values.
EventSimpleDataset	A subclass of **SimpleDataset**, it is initialized with **ChartEvent** objects, where the data is to be plotted using one of the simple plot types.
ContourDataset	A subclass of **SimpleDataset**, it adds a third dimension (z-values) to the x- and y- values of the simple dataset.

GroupDataset	Represents group data, where every x-value can have one or more y-values.
TimeGroupDataset	A subclass of **GroupDataset**, it uses **ChartCalendar** dates (a wrapper around the System.DateTime value class) as the x-values, and floating point numbers as the y-values.
ElapsedTimeGroupDataset	A subclass of **GroupDataset**, it uses **TimeSpan** objects, or milliseconds, as the x-values, and floating point numbers as the y-values.
EventGroupDataset	A subclass of **GroupDataset**, it uses **ChartEvent** objects, where the data is to be plotted using one of the group plot types.

Scale Classes

ChartScale

- **LinearScale**
- **LogScale**
- **TimeScale**
- **ElapsedTimeScale**
- **EventScale**

The **ChartScale** abstract base class defines coordinate transformation functions for a single dimension. It is useful to be able to mix and match different scale transform functions for x- and y-dimensions of the **PhysicalCoordinates** class. The job of a **ChartScale** derived object is to convert a dimension from the current *physical* coordinate system into the current *working* coordinate system.

LinearScale	A concrete implementation of the **ChartScale** class. It converts a linear physical coordinate system into the working coordinate system.
LogScale	A concrete implementation of the **ChartScale** class. It converts a logarithmic physical coordinate system into the working coordinate system.
TimeScale	A concrete implementation of the **ChartScale** class. converts a date/time physical coordinate system into the working coordinate system.
ElapsedTimeScale	A concrete implementation of the **ChartScale** class. converts an elapsed time coordinate system into the working coordinate system.

EventScale A concrete implementation of the **ChartScale** class. converts an event coordinate system into the working coordinate system.

Coordinate Transform Classes

UserCoordinates
- **WorldCoordinates**
  - **WorkingCoordinates**
    - **PhysicalCoordinates**
      - **CartesianCoordinates**
        - **ElapsedTimeCoordinates**
        - **PolarCoordinates**
        - **AntennaCoordinates**
        - **EventCoordinates**
      - **TimeCoordinates**

The coordinate transform classes maintain a 2D coordinate system. Many different coordinate systems are used to position and draw objects in a graph. Examples of some of the coordinate systems include the device coordinates of the current window, normalized coordinates for the current window and plotting area, and scaled physical coordinates of the plotting area.

UserCoordinates This contains routines for drawing lines, rectangles and text using UWP device coordinates.

WorldCoordinates This class derives from the **UserCoordinates** class and maps a device independent world coordinate system on top of the .Net device coordinate system.

WorkingCoordinates

This class derives from the **WorldCoordinates** class and extends the physical coordinate system of the *plot area* (the area typically bounded by the charts axes) to include the complete *graph area* (the area of the chart outside of the *plot area*).

PhysicalCoordinates This class is an abstract base class derived from **WorkingCoordinates** and defines the routines needed to map the physical coordinate system of a plot area into a working coordinate system. Different scale objects (**ChartScale** derived) are installed for converting physical x- and y-coordinate values into working coordinate values.

CartesianCoordinates

This class is a concrete implementation of the **PhysicalCoordinates** class and implements a coordinate system used to plot linear, logarithmic and semi-logarithmic graphs.

TimeCoordinates This class is a concrete implementation of the **PhysicalCoordinates** class and implements a coordinate system used to plot **GregorianCalenar** time-based data.

ElapsedTimeCoordinates This class is a subclass of the **CartesianCoordinates** class and implements a coordinate system used to plot elapsed time data.

PolarCoordinates This class is a subclass of the **CartesianCoordinates** class and implements a coordinate system used to plot polar coordinate data.

AntennaCoordinates This class is a subclass of the **CartesianCoordinates** class and implements a coordinate system used to plot antenna coordinate data. The antenna coordinate system differs from the more common polar coordinate system in that the radius can have plus/minus values, the angular values are in degrees, and the angular values increase in the clockwise direction.

EventCoordinates This class is a subclass of the **CartesianCoordinates** class and implements a coordinate system used to plot ChartEvent based data.

Attribute Class

ChartAttribute
ChartGradient

This class consolidates the common line and fill attributes as a single class. Most of the graph objects have a property of this class that controls the color, line thickness and fill attributes of the object. The **ChartGradient** class expands the number of color options available in the **ChartAttribute** class.

ChartAttribute This class consolidates the common line and fill attributes associated with a **GraphObj** object into a single class.

ChartGradient A **ChartGradient** can be added to a **ChartAttribute** object, defining a multicolor gradient that is applied wherever the color fill attribute is normally used

Auto-Scaling Classes

AutoScale
- **LinearAutoScale**
- **LogAutoScale**
- **TimeAutoScale**
- **ElapsedTimeAutoScale**
- **EventAutoScale**

Usually, programmers do not know in advance the scale for a chart. Normally the program needs to analyze the current data for minimum and maximum values and create a chart scale based on those values. Auto-scaling, and the creation of appropriate axes, with endpoints at even values, and well-rounded major and minor tick mark spacing, is quite complicated. The **AutoScale** classes provide tools that make automatic generation of charts easier.

AutoScale	This class is the abstract base class for the auto-scale classes.
LinearAutoScale	This class is a concrete implementation of the **AutoScale** class. It calculates scaling values based on the numeric values in **SimpleDataset** and **GroupDataset** objects. Linear scales and axes use it for auto-scale calculations.
LogAutoScale	This class is a concrete implementation of the **AutoScale** class. It calculates scaling values based on the numeric values in **SimpleDataset** and **GroupDataset** objects. Logarithmic scales and axes use it for auto-scale calculations.
TimeAutoScale	This class is a concrete implementation of the **AutoScale** class. It calculates scaling values based on the **ChartCalendar** values in **TimeSimpleDataset** and **TimeGroupDataset** objects. Date/time scales and axes use it for auto-scale calculations.
ElapsedTimeAutoScale	This class is a concrete implementation of the **AutoScale** class. It calculates scaling values based on the numeric values in **ElapsedTimeSimpleDataset** and **ElapsedTimeGroupDataset** objects. The elapsed time classes use it for auto-scale calculations.
EventAutoScale	This class is a concrete implementation of the **AutoScale**class. It calculates scaling values based on the numeric values in **EventSimpleDataset**and **EventGroupDataset**objects. The evnet classes use it for auto-scale calculations.

Chart Object Classes

Chart objects are graph objects that can be rendered in the current graph window. This is in comparison to other classes that are purely calculation classes, such as the coordinate conversion classes. All chart objects have certain information in common. This includes instances of **ChartAttribute** and **PhysicalCoordinates** classes. The **ChartAttribute** class contains basic color, line style, and gradient information for the object, while the **PhysicalCoordinates** maintains the coordinate system used by object. The majority of classes in the library derive from the **GraphObj** class, each class a specific charting object such as an axis, an axis label, a simple plot or a group plot. Add **GraphObj** derived objects (axes, plots, labels, title, etc.) to a graph using the **ChartView.AddChartObject** method.

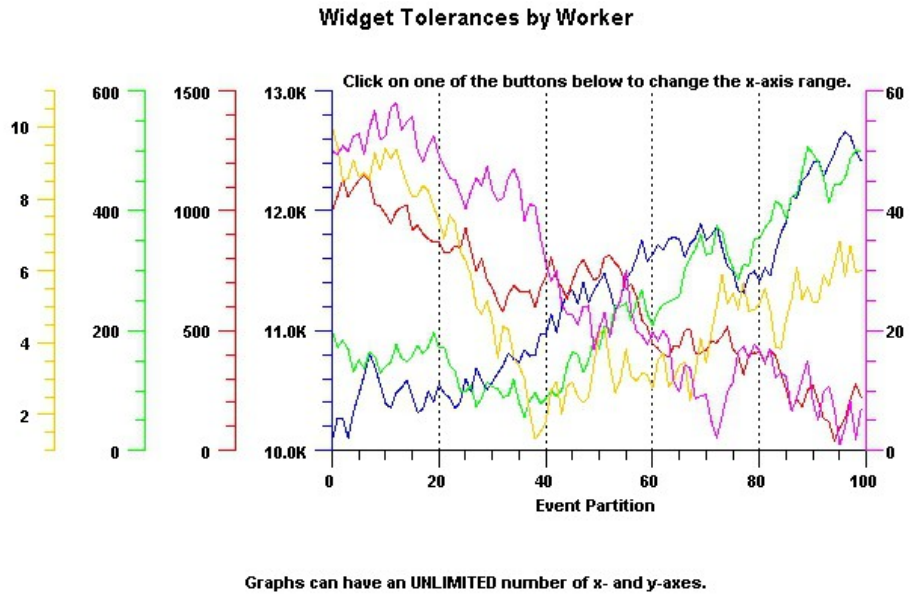
GraphObj	This class is the abstract base class for all drawable graph objects. It contains information common to all chart objects. This class includes references to instances of the **ChartAttribute** and **PhysicalCoordinates** classes. The **ChartAttribute** class contains basic color, line style, and gradient information for the object, while the **PhysicalCoordinates** maintains the coordinate system used by object. The majority of classes in the library derive from the **GraphObj** class, each class a specific charting object such as an axis, an axis label, a simple plot or a group plot
Background	This class fills the background of the entire chart, or the plot area of the chart, using a solid color, a color gradient, or a texture.

Axis Classes

Axis
- **LinearAxis**
  - **PolarAxes**
  - **AntennaAxes**
  - **ElapsedTimeAxis**
- **LogAxis**
- **TimeAxis**
- **EventAxis**

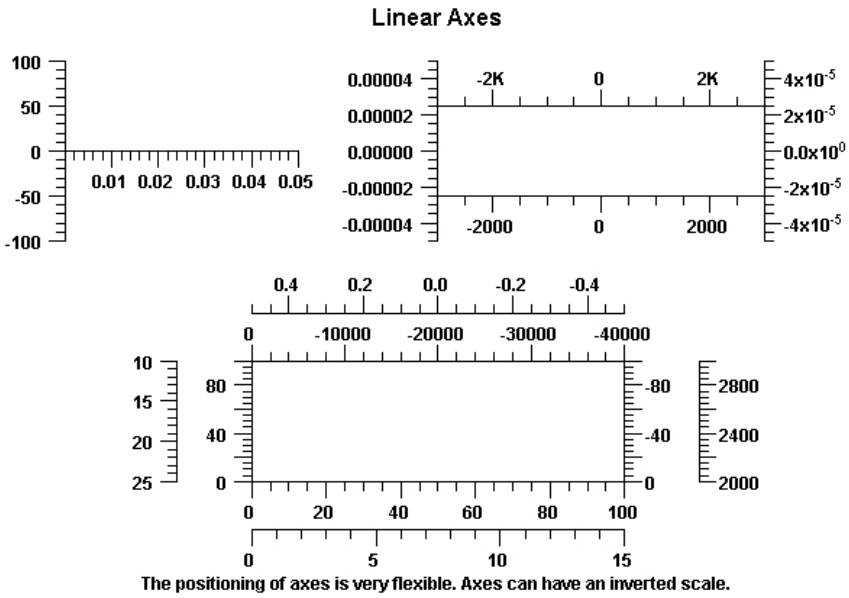
Creating a **PhysicalCoordinates** coordinate system does not automatically create a pair of x- and y-axes. Axes are separate charting objects drawn with respect to a specific **PhysicalCoordinates** object. The coordinate system and the axes do not need to have the same limits. In general, the limits of the coordinate system should be greater than or equal to the limits of the axes. The

coordinate system may have limits of 0 to 15, while you may want the axes to extend from 0 to 10.

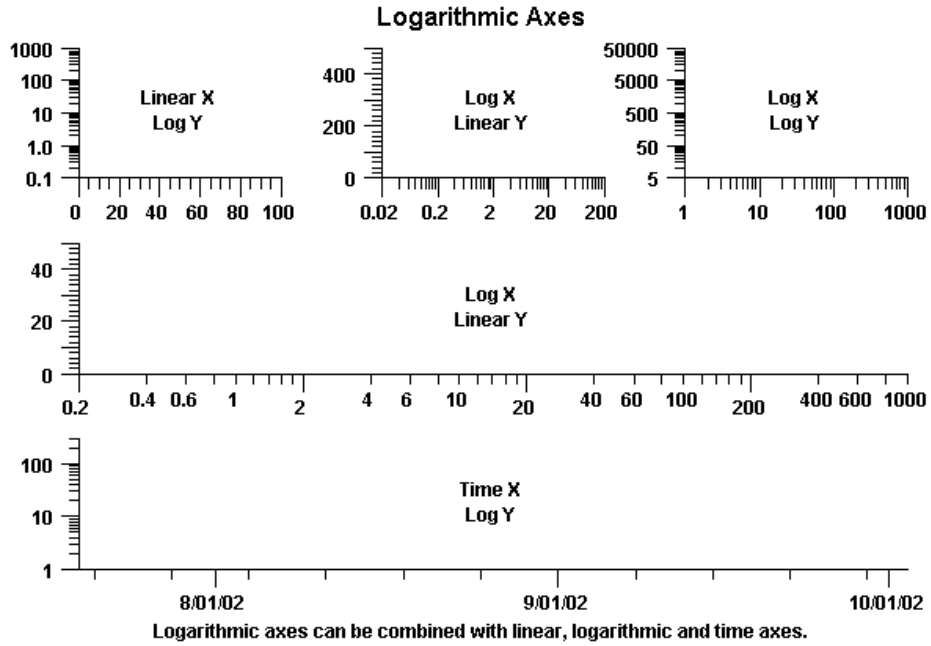


Axis

This class is the abstract base class for the other axis classes. It contains data and drawing routines common to all axis classes.

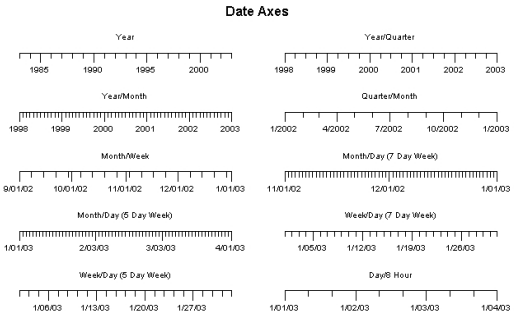


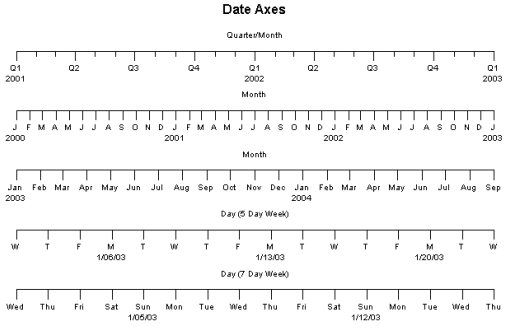
LinearAxis This class implements a linear axis with major and minor tick marks placed at equally spaced intervals.

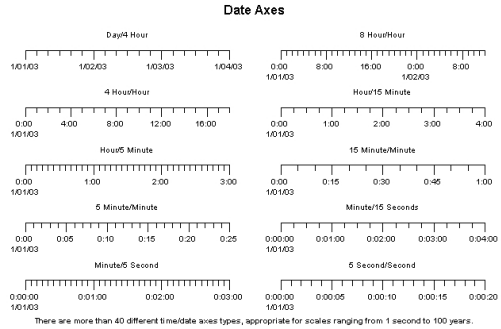


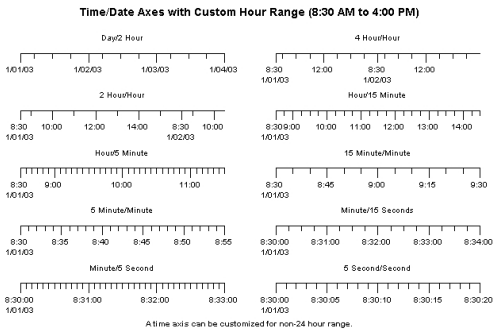
LogAxis

This class implements a logarithmic axis with major tick marks placed on logarithmic intervals, for example 1, 10,100 or 30, 300, 3000. The minor tick marks are placed within the major tick marks using linear intervals, for example 2, 3, 4, 5, 6, 7, 8, 9, 20, 30, 40, 50,.., 90. An important feature of the **LogAxis** class is that the major and minor tick marks do not have to fall on decade boundaries. A logarithmic axis must have a positive range exclusive of 0.0, and the tick marks can represent any logarithmic scale.



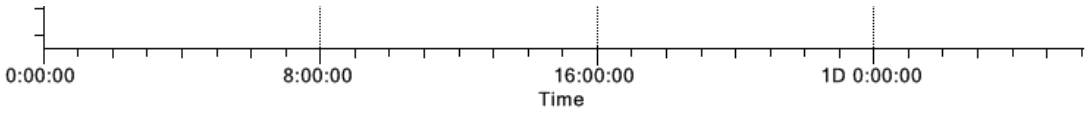






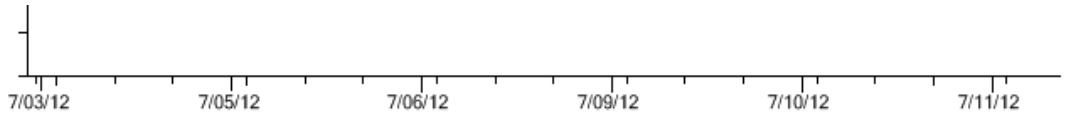
TimeAxis

This class is the most complex of the axis classes. It supports time scales ranging from 1 milliseond to hundreds of years. Dates and times are specified using the .Net **ChartCalendar** class. The major and minor tick marks can fall on any time base, where a time base represents seconds, minutes, hours, days, weeks, months or years. The scale can exclude weekends, for example, Friday, October 20, 2000 is immediately followed by Monday, October 23, 2000. A day can also have a custom range, for example a range of 9:30 AM to 4:00 PM. The chart time axis excludes time outside of this range. This makes the class very useful for the inter-day display of financial market information (stock, bonds, commodities, options, etc.) across several days, months or years.



ElapsedTimeAxis

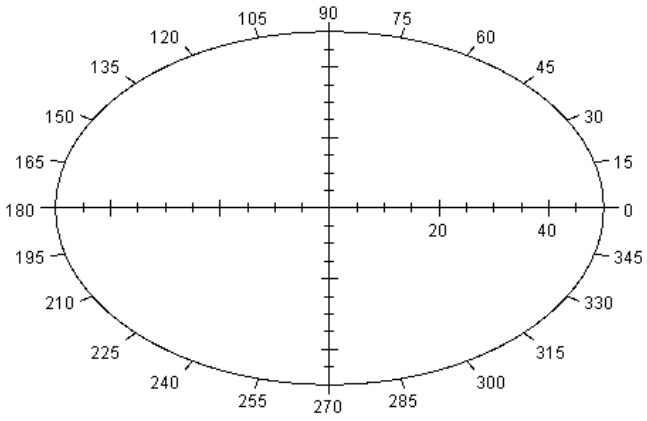
The elapsed time axis is very similar to the linear axis and is subclassed from that class. The main difference is the major and minor tick mark spacing calculated by the CalcAutoAxis method takes into account the base 60 of seconds per minute and minutes per hour, and the base 24 of hours per day. It is a continuous linear scale.



EventAxis

The event axis is a hybrid of the a time axis and the linear axis. It places major and minor tick marks on the axis, based on the event data attached to the coordinate system. Every ChartEvent object in the coordinate system does not necessarily have a tick mark, because where the data values are bunched together, this would create too many tick marks and they would overlap. Every tick mark, though, will have at least one ChartEvent object associated with it. In the case where multiple plots have ChartEvent objects with the same time stamp, a tick mark can have multiple ChartEvent objects associated with it.

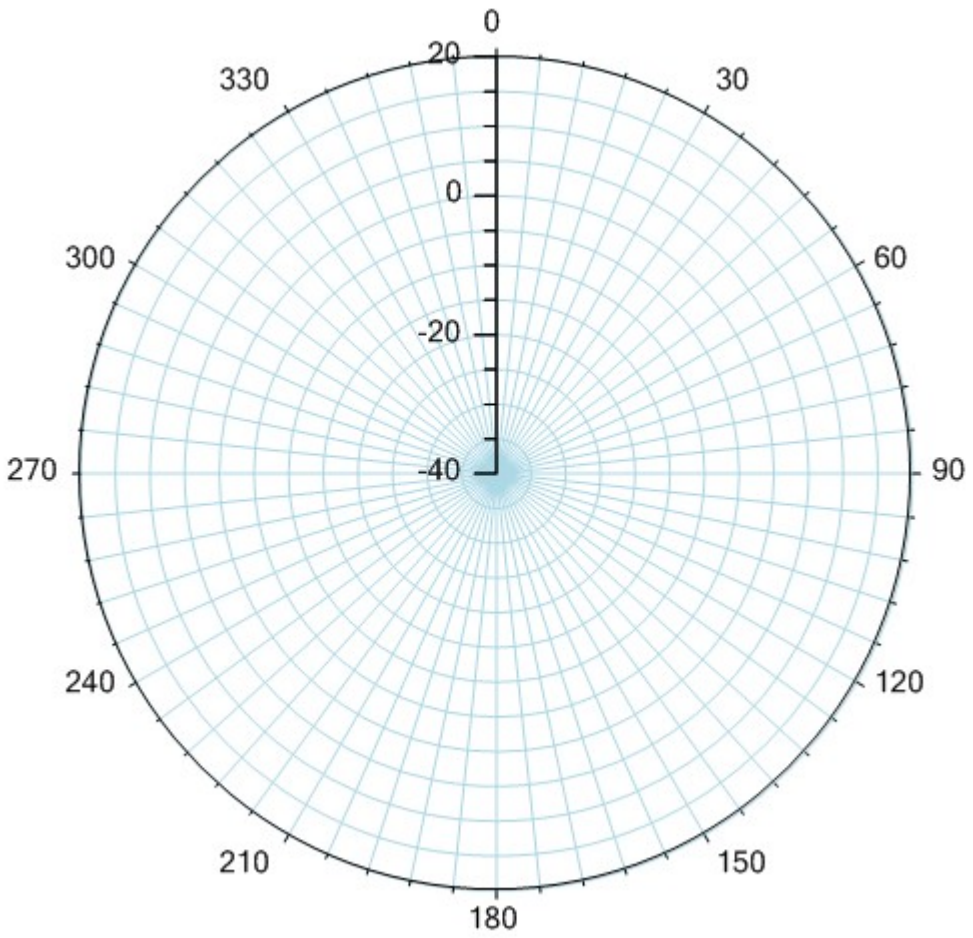
Polar Axes



A polar axis consists of the x and y axis for magnitude, and the outer circle for the angle.

PolarAxes

This class has three separate axes: two linear and one circular. The two linear axes, scaled for +- the magnitude of the polar scale, form a cross with the center of both axes at the origin (0, 0. The third axis is a circle centered on the origin with a radius equal to the magnitude of the polar scale. This circular axis represents 360 degrees (or 2 Pi radians) of the polar scale and the tick marks that circle this axis are spaced at equal degree intervals.



AntennaAxes This class has two axes: one linear y-axis and one circular axis. The linear axis is scaled for the desired range of radius values. This can extend from minus values to plus values. The second axis is a circle centered on the origin with a radius equal to the range of the radius scale. This circular axis represents 360 degrees of the antenna scale and the tick marks that circle this axis are spaced at equal degree intervals.

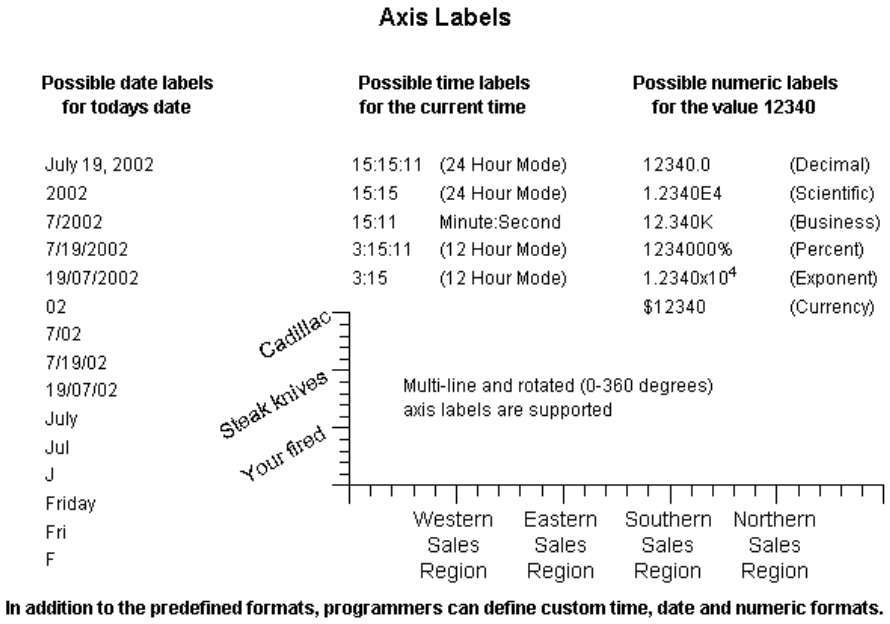
Axis Label Classes

AxisLabels

- **NumericAxisLabels**
- **StringAxisLabels**
- **PolarAxesLabels**
- **AntennaAxesLabels**
- **TimeAxisLabels**
- **ElapsedTimeAxisLabels**

EventAxisLabels

Axis labels inform the user of the x- and y-scales used in the chart. The labels center on the major tick marks of the associated axis. Axis labels are usually numbers, times, dates, or arbitrary strings.



AxisLabels — This class is the abstract base class for all axis label objects. It places numeric labels, date/time labels, or arbitrary text labels, at the major tick marks of the associated axis object. In addition to the standard font options (type, size, style, color, etc.), axis label text can be rotated 360 degrees in one degree increments.

NumericAxisLabels — This class labels the major tick marks of the **LinearAxis**, and **LogAxis** classes. The class supports many predefined and user-definable formats, including numeric, exponent, percentage, business and currency formats.

StringAxisLabels — This class labels the major tick marks of the **LinearAxis**, and **LogAxis** classes using user-defined strings.

TimeAxisLabels — This class labels the major tick marks of the associated **TimeAxis** object. The class supports many time (23:59:59) and date

(5/17/2001) formats. It is also possible to define custom date/time formats.

ElapsedTimeAxisLabels This class labels the major tick marks of the associated **ElapsedTimeAxis** object. The class supports HH:MM:SS and MM:SS formats, with decimal seconds out to 0.00001, i.e. "12:22:43.01234". It also supports a cumulative hour format (101:51:22), and a couple of day formats (4.5:51:22, 4D 5:51:22).

PolarAxesLabels This class labels the major tick marks of the associated **PolarAxes** object. The x-axis is labeled from 0.0 to the polar scale magnitude, and the circular axis is labeled counter clockwise from 0 to 360 degrees, starting at 3:00.

AntennaAxesLabels This class labels the major tick marks of the associated **AntennaAxes** object. The y-axis is labeled from the radius minimum to the radius maximum. The circular axis is labeled clockwise from 0 to 360 degrees, starting at 12:00.

EventAxisLabels This class labels the major tick marks of the associated **EventAxis** object. The class supports many time (23:59:59) and date (5/17/2001) formats. It is also possible to define custom date/time formats.

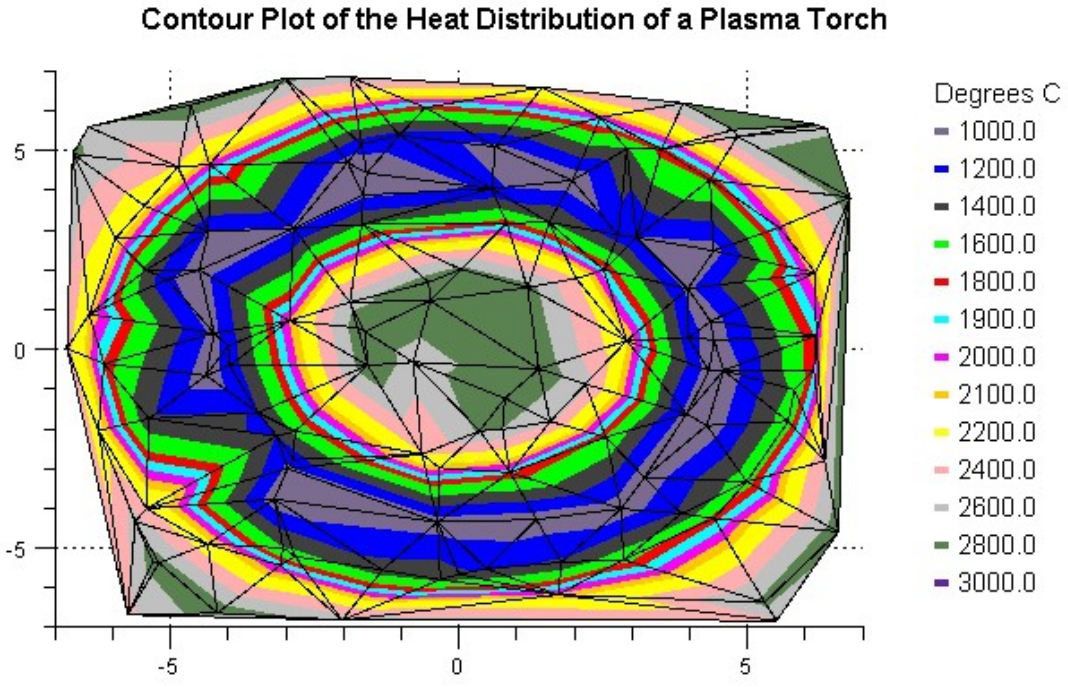
Chart Plot Classes

ChartPlot
- **ContourPlot**
- **GroupPlot**
- **PieChart**
- **PolarPlot**
- **AntennaPlot**
- **SimplePlot**

Plot objects are objects that display data organized in a **ChartDataset** class. There are six main categories: simple, group, polar, antenna, contour and pie plots. Simple plots graph data organized as a simple set of xy data points. The most common examples of simple plots are line plots, bar graphs, scatter plots and line-marker plots. Group plots graph data organized as multiple y-values for each x-value. The most common examples of group plots are stacked bar graphs, open-high-low-close plots, candlestick plots, floating stacked bar plots and "box and whisker" plots. Polar charts plot data organized as a simple set of data points, where each data point represents a polar magnitude and angle pair, rather than xy Cartesian coordinate values. The most common example of polar charts is the display of complex numbers (a + b*i*), and it is used in many engineering disciplines. Antenna charts plot data organized as a simple set of data points, where each data point represents a radius value and angle pair, rather than xy Cartesian

coordinate values. The most common example of antenna charts is the display of antenna performance and specification graphs. The contour plot type displays the iso-lines, or contours, of a 3D surface using either lines or regions of solid color. The last plot object category is the pie chart, were a pie wedge represents each data value. The size of the pie wedge is proportional to the fraction (data value / sum of all data values).

ChartPlot This class is the abstract base class for chart plot objects. It contains a reference to a **ChartDataset** derived class containing the data associated with the plot.



ContourPlot This class is a concrete implementation of the **ChartPlot** class and displays a contour plot using either lines, or regions filled with color.

Group Plot Classes

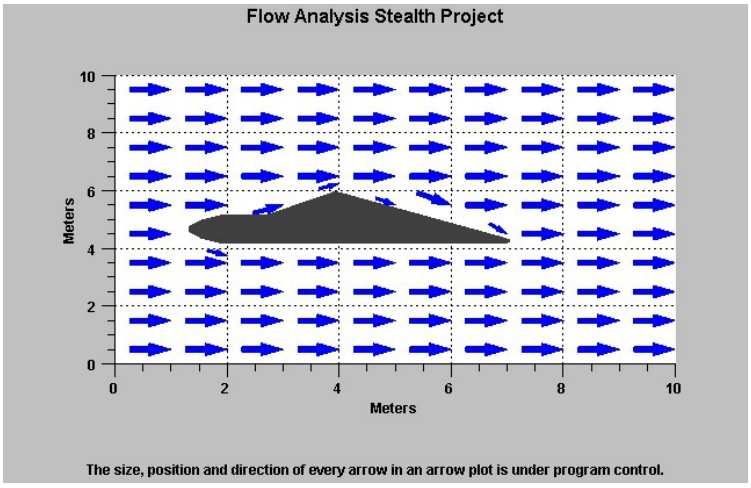
GroupPlot

- **ArrowPlot**
- **BoxWhiskerPlot**

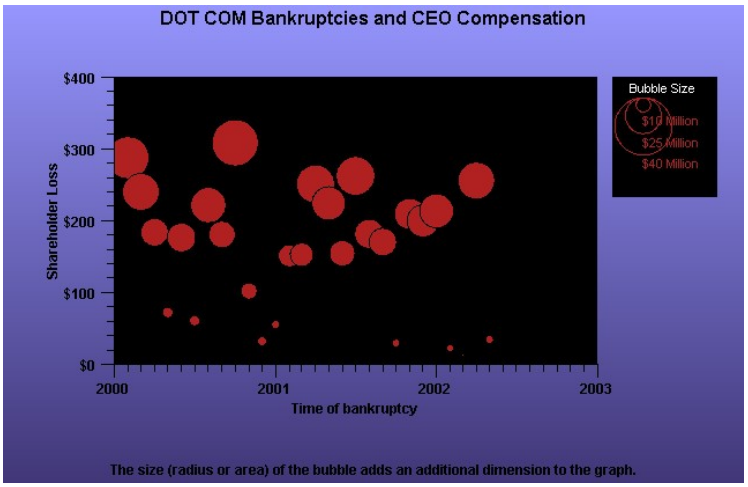
BubblePlot
CandlestickPlot
CellPlot
ErrorBarPlot
FloatingBarPlot
FloatingStackedBarPlot
GroupBarPlot
GroupVersaPlot
HistogramPlot
LineGapPlot
MultiLinePlot
OHLCPlot
StackedBarPlot
StackedLinePlot
GroupVersaPlot

Group plots use data organized as arrays of x- and y-values, where there is one or more y for every x.. Group plot types include multi-line plots, stacked line plots, stacked bar plots, group bar plots, error bar plots, floating bar plots, floating stacked bar plots, open-high-low-close plots, candlestick plots, arrow plots, histogram plots, cell plots, “box and whisker” plots, and bubble plots.

GroupPlot This class is an abstract base class for all group plot classes.

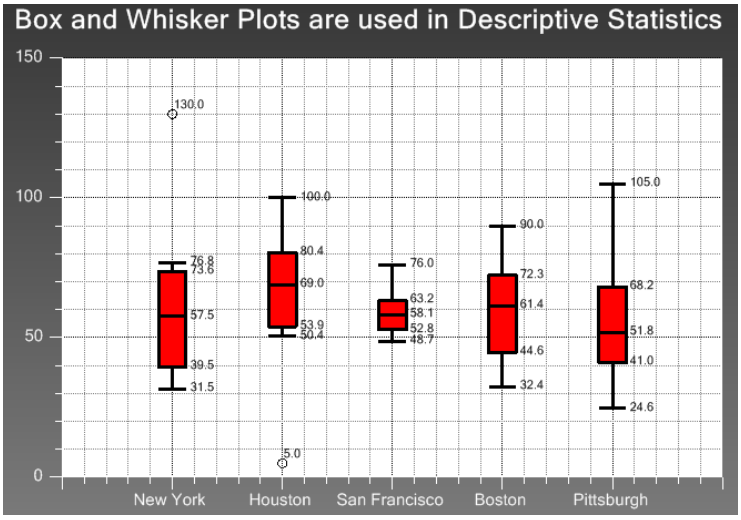


ArrowPlot This class is a concrete implementation of the **GroupPlot** class and it displays a collection of arrows as defined by the data in a group dataset. The position, size, and rotation of each arrow in the collection is independently controlled



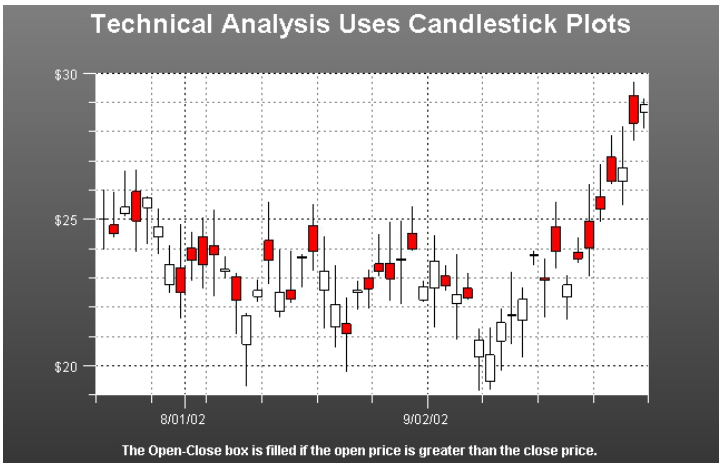
BubblePlot

This class is a concrete implementation of the **GroupPlot** class and displays bubble plots. The values in the dataset specify the position and size of each bubble in a bubble chart.

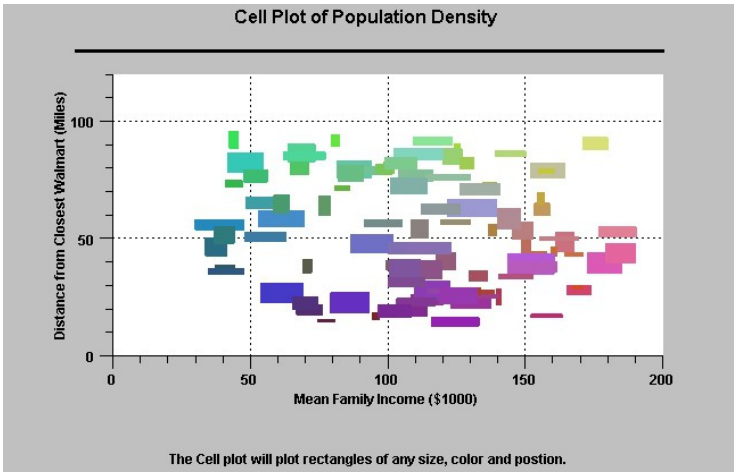


BoxWhiskerPlot

This class is a concrete implementation of the **GroupPlot** class and displays box and whisker plots. The BoxWhiskerPlot class graphically depicts groups of numerical data through their five-number summaries (the smallest observation, lower quartile (Q1), median (Q2), upper quartile (Q3), and largest observation).

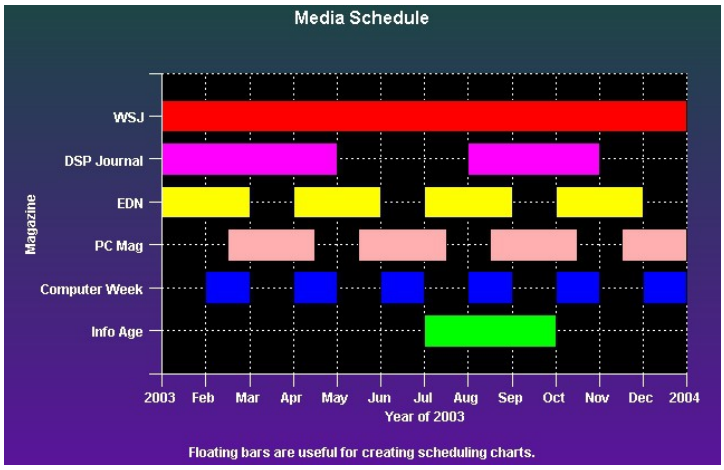


CandlestickPlot This class is a concrete implementation of the **GroupPlot** class and displays stock market data in an open-high-low-close format common in financial technical analysis.

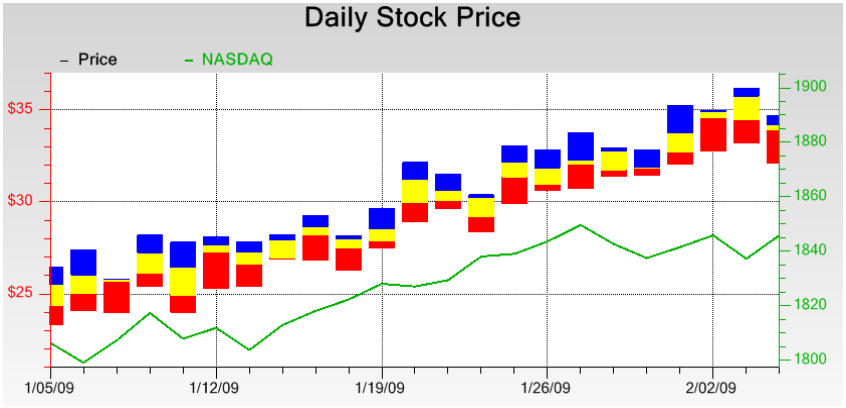


CellPlot This class is a concrete implementation of the **GroupPlot** class and displays cell plots. A cell plot is a collection of rectangular objects with independent positions, widths and heights, specified using the values of the associated group dataset.

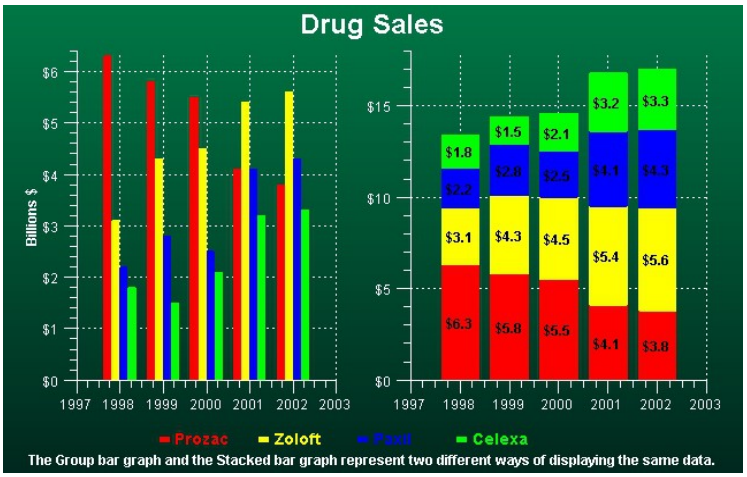
ErrorBarPlot This class is a concrete implementation of the **GroupPlot** class and displays error bars. Error bars are two lines positioned about a data point that signify the statistical error associated with the data point



FloatingBarPlot This class is a concrete implementation of the **GroupPlot** class and displays free-floating bars in a graph. The bars are free floating because each bar does not reference a fixed base value, as do simple bar plots, stacked bar plots and group bar plots.



FloatingStackedBarPlot This class is a concrete implementation of the **GroupPlot** class and displays free-floating stacked bars. The bars are free floating because each bar does not reference a fixed base value, as do simple bar plots, stacked bar plots and group bar plots.



GroupBarPlot

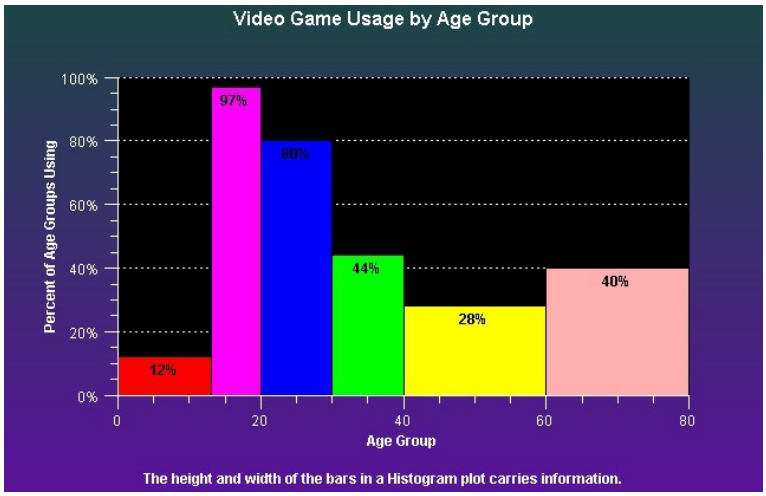
This class is a concrete implementation of the **GroupPlot** class and displays group data in a group bar format. Individual bars, the height of which corresponds to the group y-values of the dataset, display side by side, as a group, justified with respect to the x-position value for each group. The group bars share a common base value.

StackedBarPlot

This class is a concrete implementation of the **GroupPlot** class and displays data as stacked bars. In a stacked bar plot each group is stacked on top of one another, each group bar a cumulative sum of the related group items before it.

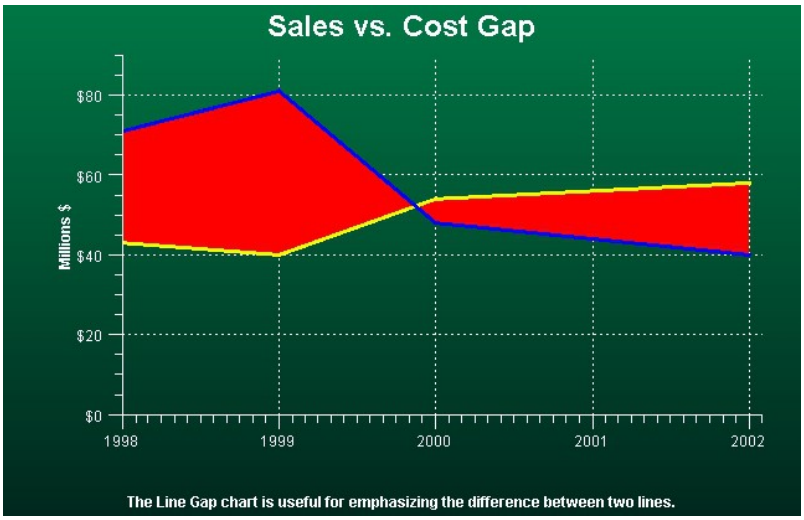
GroupVersaPlot

The **GroupVersaPlot** is a plot type that can be any of the eight group plot types: GROUPBAR, STACKEDBAR, CANDLESTICK, OHLC, MULTILINE, STACKEDLINE, FLOATINGBAR and FLOATING_STACKED_BAR. Use it when you want to be able to change from one plot type to another, without deleting the instance of the old plot object and creating an instance of the new.



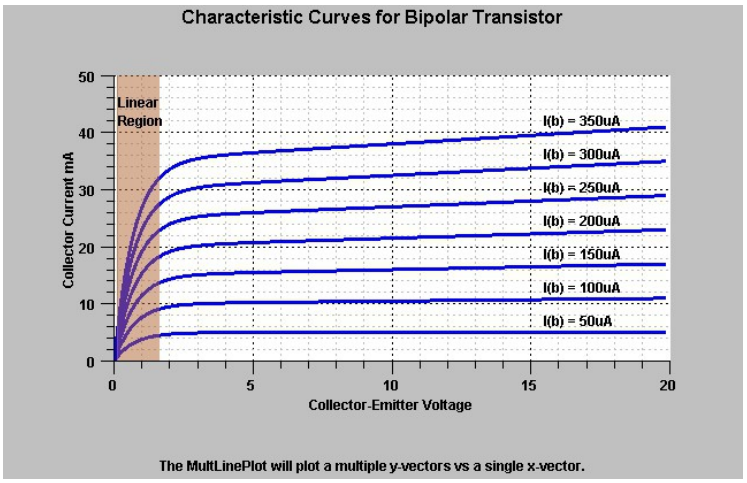
HistogramPlot

This class is a concrete implementation of the **GroupPlot** class and displays histogram plots. A histogram plot is a collection of rectangular objects with independent widths and heights, specified using the values of the associated group dataset. The histogram bars share a common base value.



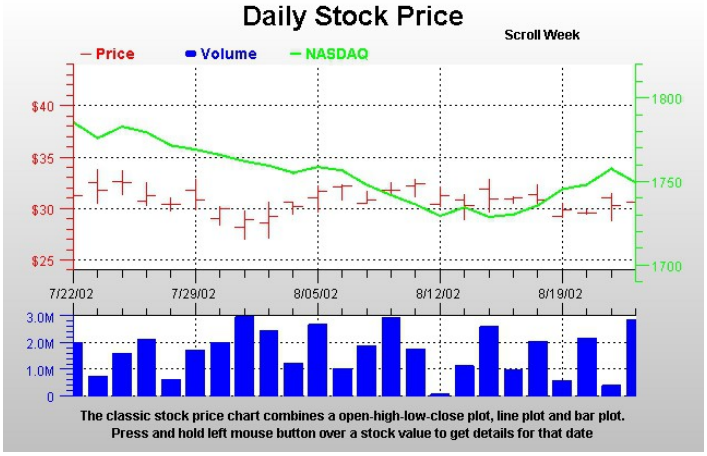
LineGapPlot

This class is a concrete implementation of the **GroupPlot** class. A line gap chart consists of two lines plots where a contrasting color fills the area between the two lines, highlighting the difference.



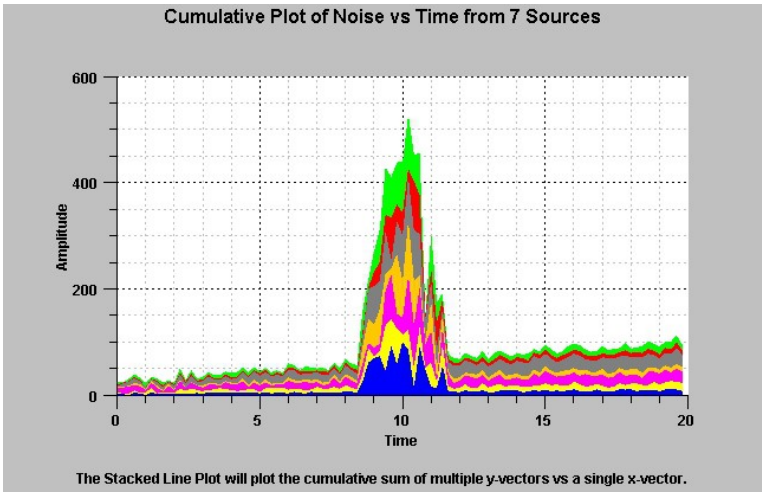
MultiLinePlot

This class is a concrete implementation of the **GroupPlot** class and displays group data in multi-line format. A group dataset with four groups will display four separate line plots. The y-values for each line of the line plot represent the y-values for each group of the group dataset. Each line plot share the same x-values of the group dataset.



OHLCPlot

This class is a concrete implementation of the **GroupPlot** class and displays stock market data in an open-high-low-close format common in financial technical analysis. Every item of the plot is a vertical line, representing High and Low values, with two small horizontal "flags", one left and one right extending from the vertical High-Low line and representing the Open and Close values.



StackedLinePlot This class is a concrete implementation of the **GroupPlot** class and displays data in a stacked line format. In a stacked line plot each group is stacked on top of one another, each group line a cumulative sum of the related group items before it.

Polar Plot Classes

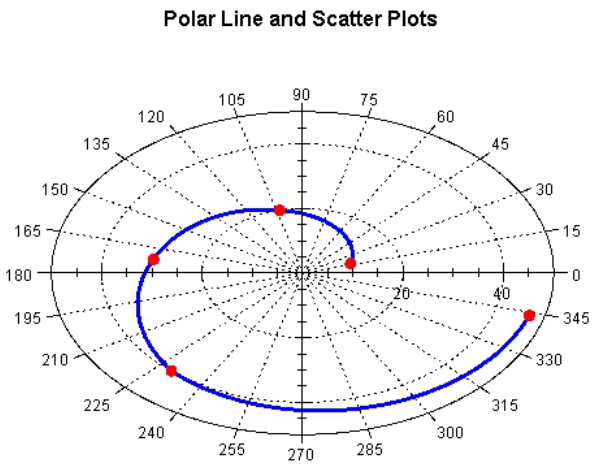
PolarPlot

PolarLinePlot

PolarScatterPlot

Polar plots that use data organized as arrays of x- and y-values, where an x-value represents the magnitude of a point in polar coordinates, and the y-value represents the angle, in radians, of a point in polar coordinates. Polar plot types include line plots and scatter plots.

PolarPlot This class is an abstract base class for the polar plot classes.



PolarLinePlot This class is a concrete implementation of the **PolarPlot** class and displays data in a simple line plot format. The lines drawn between adjacent data points use polar coordinate interpolation.

PolarScatterPlot This class is a concrete implementation of the **PolarPlot** class and displays data in a simple scatter plot format.

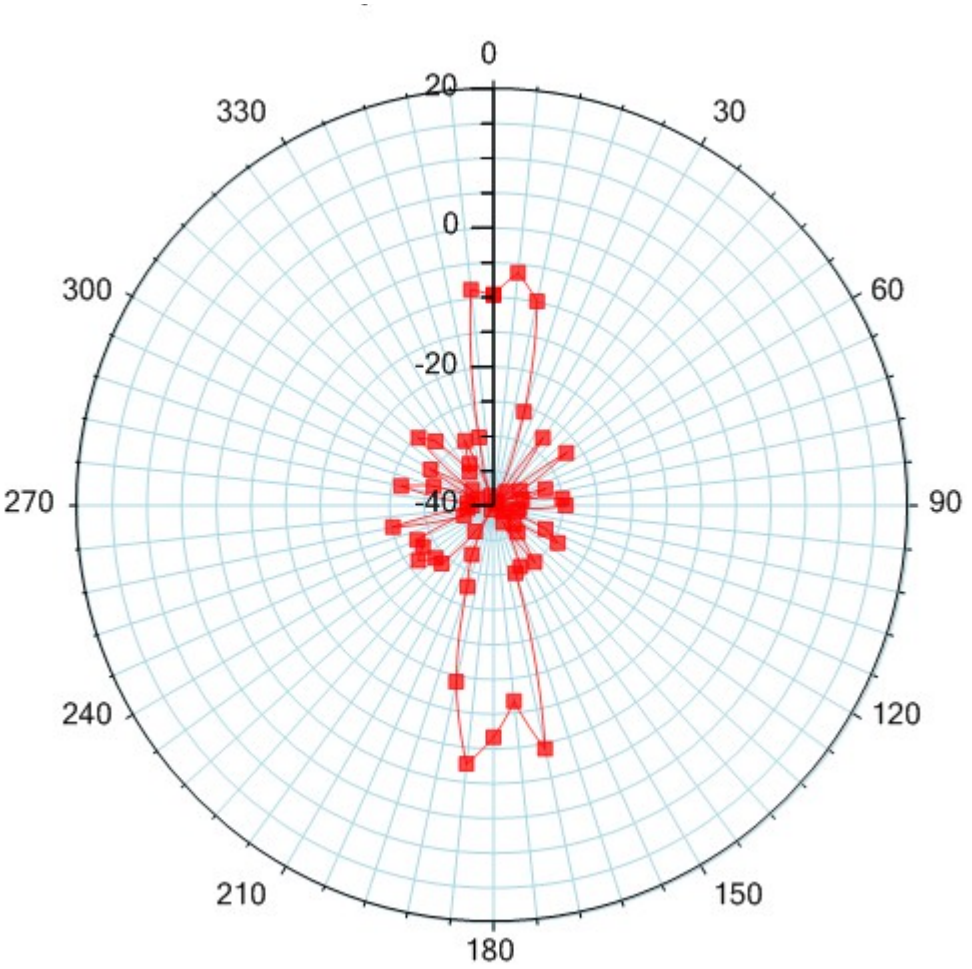
Antenna Plot Classes

AntennaPlot
- **AntennaLinePlot**
- **AntennaScatterPlot**
- **AntennaLineMarkerPlot**

GraphObj
- **AntennaAnnotation**

Antenna plots that use data organized as arrays of x- and y-values, where an x-value represents the radial value of a point in antenna coordinates, and the y-value represents the angle, in degrees, of a point in antenna coordinates. Antenna plot types include line plots, scatter plots, line marker plots, and an annotation class.

AntennaPlot This class is an abstract base class for the polar plot classes.



AntennaLineMarkerPlot

AntennaLinePlot This class is a concrete implementation of the **AntennaPlot** class and displays data in a simple line plot format. The lines drawn between adjacent data points use antenna coordinate interpolation.

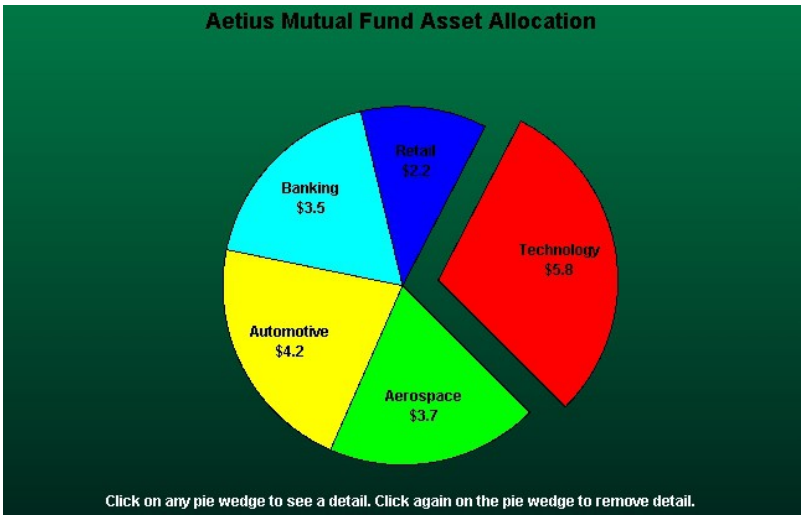
AntennaScatterPlot This class is a concrete implementation of the **AntennaPlot** class and displays data in a simple scatter plot format.

AntennaLineMarkerPlot This class is a concrete implementation of the **AntennaPlot** class and displays data in a simple line markder plot format.

AntennaAnnotation This class is used to highlight, or mark, a specific attribute of the chart. It can mark a constant radial value using a circle, or it can mark a constant angular value using a radial line from the origin to the outer edge of the scale.

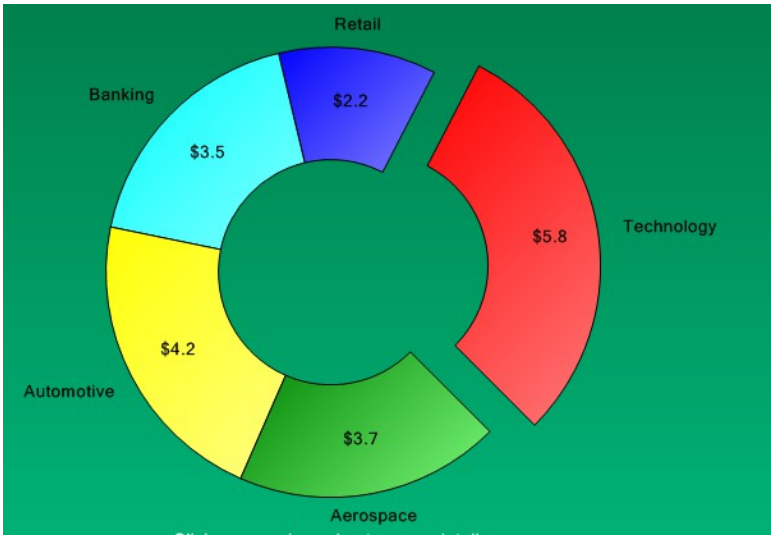
Pie and Ring Chart Classes

It uses data organized as arrays of x- and y-values, where an x-value represents the numeric value of a pie wedge, and a y-value specifies the offset (or "explosion") of a pie wedge with respect to the center of the pie.



PieChart

The pie chart plots data in a simple pie chart format. It uses data organized as arrays of x- and y-values, where an x-value represents the numeric value of a pie wedge, and a y-value specifies the offset (or "explosion") of a pie wedge with respect to the center of the pie.



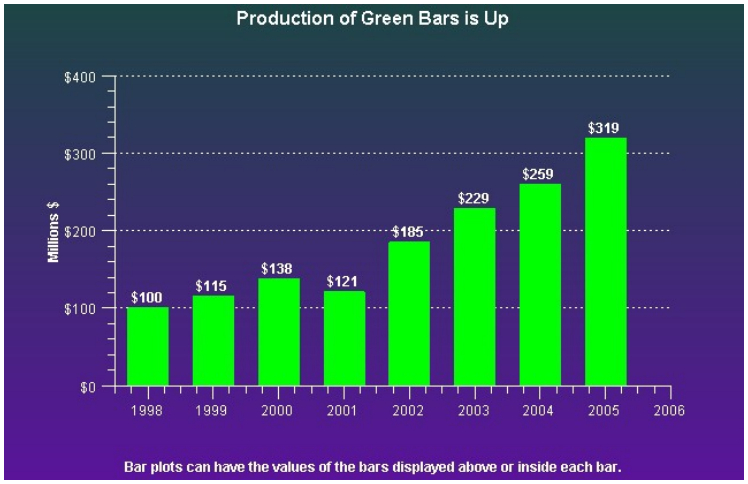
RingChart The ring chart plots data in a modified pie chart format known as a ring chartt. It uses data organized as arrays of x- and y-values, where an x-value represents the numeric value of a ring segment, and a y-value specifies the offset (or "explosion") of a ring segment with respect to the origin of the ring.

Simple Plot Classes

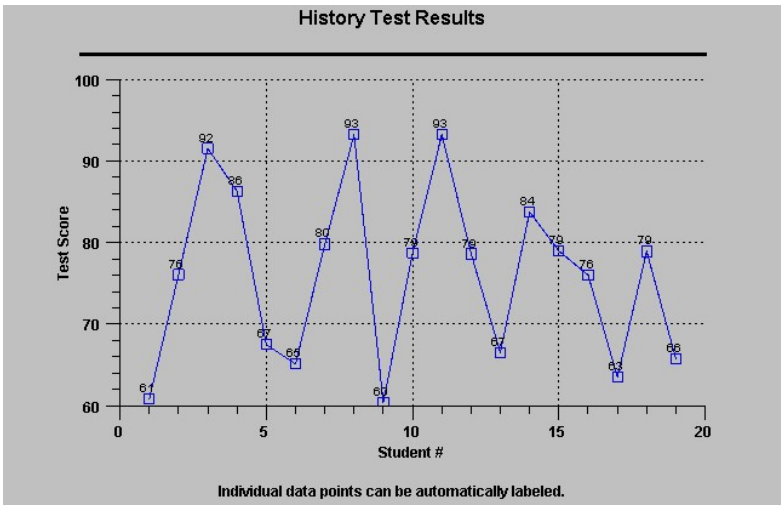
SimplePlot
- **SimpleBarPlot**
- **SimpleLineMarkerPlot**
- **SimpleLinePlot**
- **SimpleScatterPlot**
- **SimpleVeraPlot**

Simple plots use data organized as a simple array of xy points, where there is one y for every x. Simple plot types include line plots, scatter plots, bar graphs, and line-marker plots.

SimplePlot This class is an abstract base class for all simple plot classes.

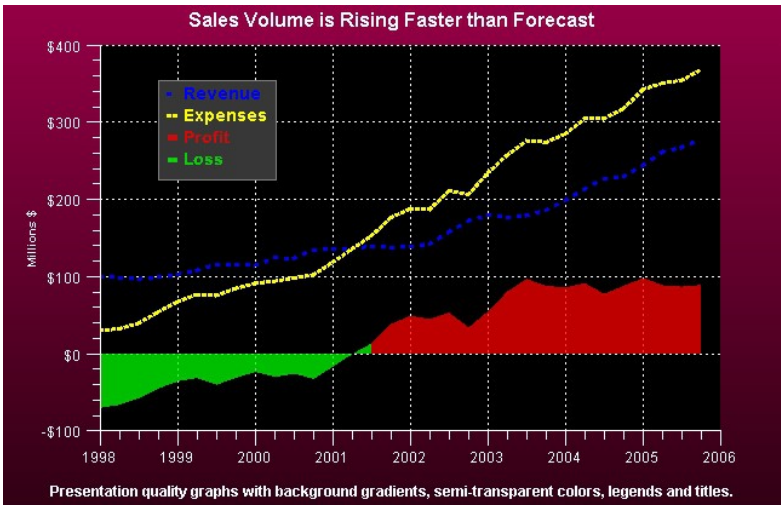


SimpleBarPlot This class is a concrete implementation of the **SimplePlot** class and displays data in a bar format. Individual bars, the maximum value of which corresponds to the y-values of the dataset, are justified with respect to the x-values.



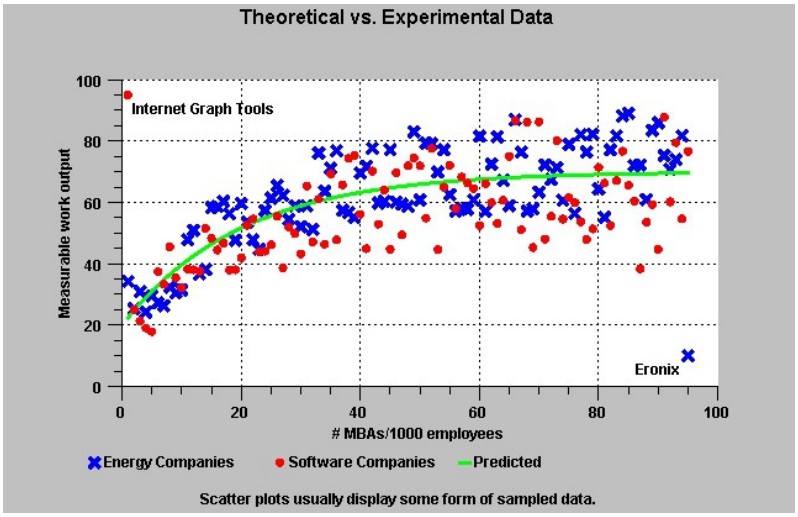
SimpleLineMarkerPlot

This class is a concrete implementation of the **SimplePlot** class and it displays simple datasets in a line plot format where scatter plot symbols highlight individual data points.



SimpleLinePlot

This class is a concrete implementation of the **SimplePlot** class it displays simple datasets in a line plot format. Adjacent data points are connected using a straight, or a step line.



SimpleScatterPlot This class is a concrete implementation of the **SimplePlot** class and it displays simple datasets in a scatter plot format where each data point is represented using a symbol.

SimpleVersaPlot The **SimpleVersaPlot** is a plot type that can be any of the four simple plot types: LINE_MARKER_PLOT, LINE_PLOT, BAR_PLOT, SCATTER_PLOT. It is used when you want to be able to change from one plot type to another, without deleting the instance of the old plot object and creating an instance of the new.

Legend Classes

LegendItem
BubblePlotLegendItem
Legend
- **StandardLegend**
- **BubblePlotLegend**

Legends provide a key for interpreting the various plot objects in a graph. It organizes a collection of legend items, one for each plot objects in the graph, and displays them in a rectangular frame.

Legend This class is the abstract base class for chart legends.

LegendItem This class is the legend item class for all plot objects except for bubble plots. Each legend item manages one symbol and

descriptive text for that symbol. The **StandardLegend** class uses objects of this type as legend items.

BubblePlotLegendItem This class is the legend item class for bubble plots. Each legend item manages a circle and descriptive text specifying the value of a bubble of this size. The **BubblePlotLegend** class uses objects of this type as legend items.

StandardLegend This class is a concrete implementation of the **Legend** class and it is the legend class for all plot objects except for bubble plots. The legend item objects display in a row or column format. Each legend item contains a symbol and a descriptive string. The symbol normally associates the legend item to a particular plot object, and the descriptive string describes what the plot object represents.

BubblePlotLegend This class is a concrete implementation of the **Legend** class and it is a legend class used exclusively with bubble plots. The legend item objects display as offset, concentric circles with descriptive text giving the key for the value associated with a bubble of this size.

ChartGrid Classes

ChartGrid
- **PolarGrid**
- **AntennaGrid**

Grid lines are perpendicular to an axis, extending the major and/or minor tick marks of the axis across the width or height of the plot area of the chart.

ChartGrid This class defines the grid lines associated with an axis. Grid lines are perpendicular to an axis, extending the major and/or minor tick marks of the axis across the width or height of the plot area of the chart. This class works in conjunction with the **LinearAxis**, **LogAxis** and **TimeAxis** classes.

PolarGrid This class defines the grid lines associated with a polar axis. A polar chart grid consists of two sets of lines. The first set is a group of concentric circles, centered on the origin and passing through the major and/or minor tick marks of the polar magnitude horizontal and vertical axes. The second set is a group of radial lines, starting at the origin and extending to the outermost edge of the polar plot circle, passing through the major and minor tick

marks of the polar angle circular axis. This class works in conjunction with the **PolarAxes** class.

AntennaGrid — Analogous to the **PolarGrid**, this class draws radial, and circular grid lines for an Antenna chart.

Chart Text Classes

ChartText
- **ChartTitle**
- **AxisTitle**
- **ChartLabel**
  - **NumericLabel**
  - **TimeLabel**
  - **StringLabel**
  - **ElapsedTimeLabel**

The chart text classes draw one or more strings in the chart window. Different classes support different numeric formats, including floating point numbers, date/time values and multi-line text strings. International formats for floating point numbers and date/time values are also supported.

ChartText — This class draws a string in the current chart window. It is the base class for the **ChartTitle**, **AxisTitle** and **ChartLabel** classes. The **ChartText** class also creates independent text objects. Other classes that display text also use it internally.

ChartTitle — This class displays a text string as the title or footer of the chart.

AxisTitle — This class displays a text string as the title for an axis. The axis title position is outside of the axis label area. Axis titles for y-axes are rotated 90 degrees.

ChartLabel — This class is the abstract base class of labels that require special formatting.

NumericLabel — This class is a concrete implementation of the **ChartLabel** class and it displays formatted numeric values.

TimeLabel — This class is a concrete implementation of the **ChartLabel** class and it displays formatted **ChartCalendar** dates.

ElapsedTimeLabel	This class is a concrete implementation of the **ChartLabel** class and it displays numeric values formatted as elapsed time strings **(12:32:21).**
StringLabel	This class is a concrete implementation of the **ChartLabel** class that formats string values for use as axis labels.

Miscellaneous Chart Classes

Marker
ChartImage
ChartShape
ChartSymbol

Various classes are used to position and draw objects that can be used as standalone objects in a graph, or as elements of other plot objects.

Marker	This class displays one of five marker types in a graph. The marker is used to create data cursors, or to mark data points.
ChartImage	This class encapsulates a class, defining a rectangle in chart coordinates that the image is placed in JPEG and other image files can be imported using the WriteableBitmap class and displayed in a chart.
ChartShape	This class encapsulates a **Windows.UI.Xaml.Media.PathGeometry** class, placing the shape in a chart using a position defined in chart coordinates. A chart can display any object that can be defined using **PathGeometry** class.
ChartSymbol	This class defines symbols used by the **SimplePlot** scatter plot functions. Pre-defined symbols include square, triangle, diamond, cross, plus, star, line, horizontal bar, vertical bar, 3D bar and circle.

Mouse Interaction Classes

MouseListener
- **MoveObj**
- **FindObj**

DataToolTip
DataCursor
    MoveData
MagniView
MoveCoordinates
MultiMouseListener
ChartZoom

Several classes implement delegates for mouse events. The **MouseListener** class implements a generic interface for managing mouse events in a graph window. The **DataCursor**, **MoveData**, **MoveObj, ChartZoom, MagniView and MoveCoordinates** classes also implement mouse event delegates that use the mouse to mark, move and zoom chart objects and data.

MouseListener	This class implements .Net delegates that trap generic mouse events (button events and mouse motion events) that take place in a **ChartView** window. A programmer can derive a class from **MouseListener** and override the methods for mouse events, creating a custom version of the class.
MoveObj	This class extends the **MouseListener** class and it can select chart objects and move them. Moveable chart objects include axes, axes labels, titles, legends, arbitrary text, shapes and images. Use the **MoveData** class to move objects derived from **SimplePlot**.
FindObj	This class extends the **MouseListener** class, providing additional methods that selectively determine what graphical objects intersect the mouse cursor.
DataCursor	This class combines the **MouseListener** class and **Marker** class. Press a mouse button and the selected data cursor (horizontal and/or vertical line, cross hairs, or a small box) appears at the point of the mouse cursor. The data cursor tracks the mouse motion as long as the mouse button is pressed. Release the button and the data cursor disappears. This makes it easier to line up the mouse position with the tick marks of an axis.
MoveData	This class selects and moves individual data points of an object derived from the **SimplePlot** class.
DataToolTip	A data tooltip is a popup box that displays the value of a data point in a chart. The data value can consist of the x-value, the y-value, x- and y-values, group values and open-high-low-close values, for a given point in a chart.

ChartZoom This class implements mouse controlled zooming for one or more simultaneous axes. The user starts zooming by holding down a mouse button with the mouse cursor in the plot area of a graph. The mouse is dragged and then released. The rectangle established by mouse start and stop points defines the new, zoomed, scale of the associated axes. Zooming has many different modes. Some of the combinations are:

- One x or one y axis
- One x and one y axes
- One x and multiple y axes
- One y and multiple x axes
- Multiple x and y axes

MagniView This class implements mouse controlled magnification for one or more simultaneous axes. This class implements a chart magnify class based on the **MouseListener** class. It uses two charts; the source chart and the target chart. The source chart displays the chart in its unmagnified state. The target chart displays the chart in the magnified state. The mouse positions a **MagniView** rectangle within the source chart, and the target chart is re-scaled and redrawn to match the extents of the **MagniView** rectangle from the source chart.

MoveCoordinates This class extends the **MouseListener** class and it can move the coordinate system of the underlying chart, analogous to moving (chaging the coordinates of) an internet map by "grabbing" it with the mouse and dragging it.

MultiMouseListener This class is used by the **ChartView** class to support multiple mouse listeners at the same time.

File and Printer Rendering Classes

ChartPrint
BufferedImage

ChartPrint This class implements printing using the Windows App **Windows.UI.Xaml.Printing** and **Windows.Graphics.Printing**

print-related services. It can select, setup, and output a chart to a printer.

BufferedImage This class will convert a ChartView object to a **Windows.UI.Xaml.Media.Imaging** RenderTargetBitmap object. Optionally, the class saves the buffered image to an image file.

Miscellaneous Utility Classes

ChartCalendar
CSV
Dimension
Point2D
GroupPoint2D
DoubleArray
DoubleArray2D
BoolArray
Point3D
NearestPointData
TickMark
Polysurface
Rectangle2D

ChartCalendar This class contains utility routines used to process **ChartCalendar** date objects.

CSV This is a utility class for reading and writing CSV (Comma Separated Values) files.

Dimension This is a utility class for handling dimension (height and width) information using doubles, rather than the integers used by the Size class.

Point2D This class encapsulates an xy pair of values as doubles (more useful in this software than the .Net Point and PointF classes.

GroupPoint2D This class encapsulates an x-value, and an array of y-values, representing the x and y values of one column of a group data set.

DoubleArray This class is used as an alternative to the standard .Net Array class, adding routines for resizing of the array, and the insertion and deletion of double based data elements.

DoubleArray2D	This class is used as an alternative to the standard .Net 2D Array class, adding routines for resizing of the array, and the insertion and deletion of double based data elements.
BoolArray	This class is used as an alternative to the standard .Net Array class, adding routines for resizing of the array, and the insertion and deletion of bool based data elements.
Point3D	This class encapsulates an xyz set of double values used to specify 3D data values.
NearestPointData	This is a utility class for returning data that results from nearest point calculations.
TickMark	The axis classes use this class to to organize the location of the individual tick marks of an axis.
Polysurface	This is a utility class that defines complex 3D shapes as a list of simple 3-sided polygons. The contour plotting routines use it.
Rectangle2D	This is a utility class that extends the RectangleF class, using doubles as internal storage.

A diagram depicts the class hierarchy of the QCChart2D for Windows Apps library.

ChartObj → Rectangle2D, Point2D, Point3D, GroupPoint2D, DoubleArray, DoubleArray2D, BoolArray, Polysurface

- Arrow
- ChartCalendar
- ChartEvent
- CSV
- Dimension
- Point3D
- NearestPointData
- Polysurface
- ChartScale
  - LinearScale
  - LogScale
  - TimeScale
  - ElapsedTimeScale
  - EventScale
- UserCoordinates
  - WorldCoordinates
    - WorkingCoordinates
      - PhysicalCoordinates
        - CartesianCoordinates
          - PolarCoordinates
          - AntennaCoordinates
          - EventCoordiinates
        - TimeCoordinates
        - ElapsedTimeCoordinates
- ChartDataset
  - SimpleDataset
  - SimpleDataset
    - TimeSimpleDataset
    - ElapsedTimeSimpleDataset
    - ContourDataset
    - EventSimpleDataset
  - GroupDataset
    - TimeGroupDataset
    - ElapsedTimeGroupDataset
    - EventGroupDataset
- AutoScale
  - LinearAutoScale
  - LogAutoScale
  - TimeAutoScale
  - ElapsedTimeAutoScale
  - EventAutoScale
- MouseListener
  - MoveObj
  - FindObj
  - DataToolTip
  - ChartZoom
  - MagniView
  - MoveCoordinates
  - MultiMouseListener

DataCursor
- MoveData

ChartAttribute
ChartGradient
ChartPrint
BufferedImage

Windows.UI.Xaml.Controls.UserControl
ChartView

- GraphObj
  - AntennaAnnotation
  - TickMark
  - Axis
    - LinearAxis
      - PolarAxes
      - AntennaAxes
    - LogAxis
    - TimeAxis
    - ElapsedTimeAxis
    - EventAxis
  - ChartText
    - ChartTitle
    - AxisTitle
    - ChartLabel
      - NumericLabel
        - BarDatapointValue
      - TimeLabel
      - ElapsedTimeLabel
      - StringLabel
    - AxisLabels
      - NumericAxisLabels
      - TimeAxisLabels
      - ElapsedTimeAxisLabels
      - StringAxisLabels
      - PolarAxesLabels
      - AntennaAxesLabels
      - EventAxisLabels
  - ChartGrid
    - PolarGrid
    - AntennaGrid
  - LegendItem
  - BubblePlotLegendItem
  - Legend
    - StandardLegend
    - BubblePlotLegend
  - ChartPlot
    - SimplePlot
      - SimpleLinePlot
      - SimpleBarPlot
      - SimpleScatterPlot
      - SimpleLineMarkerPlot
      - SimpleVersaPlot
    - GroupPlot
      - ArrowPlot
      - BubblePlot
      - CandlestickPlot
      - CellPlot
      - ErrorBarPlot
      - FloatingBarPlot
      - FloatingStackedBarPlot
      - GroupBarPlot
      - HistogramPlot
      - LineGapPlot
      - MultiLinePlot
      - OHLCPlot
      - StackedBarPlot
      - StackedLinePlot
      - BoxWhiskerPlot
      - GroupVersaPlot
    - PieChart
    - RingChart
    - PolarPlot
      - PolarLinePlot
      - PolarScatterPlot
    - AntennaPlot
      - AntennaLinePlot
      - AntennaScatterPlot
      - AntennaLineMarkerPlot
- Background
- ChartImage
- ChartShape
- ChartSymbol
- Marker
- ChartZoom

Source Code Differences between the .Net Forms version of QCChart2D/QCRTGraph and the UWP version.

There are some minor difference in the names of classes between this, and the original .Net Forms based version of QCChart2D. These differences are summarized below. All of these changes are the result of changes in base types used by UWP, or conflicts between the original QCChart2D software and base classes in UWP. If you intend to translate applications from the original QCChart2D for .Net software to the UWP version, take special note of these differences.

It can be confusing, because in UWP, many classes have the same name as their .Net Forms equivalents: **Color**, **Timer**, **Color.FromArgb** and **MouseEventArgs** are a few. It is just that they are in different namespaces. Usually the default collection of using statements at the top of UWP classes point you to the correct namespaces. Other UWP classes have slightly different names than their .Net equivalents: **Colors**, **DashStyles**, **FontStyle**, **FontWeights**, **Color.FromRgb**, **MouseButtons** and **MouseButtonEventArgs**.

Color

The color class used in the .Net Forms based version of QCChart2D is derived from the **System.Drawing.Color** class. The UWP version of the **Color** class derives from the **Windows.UI.Color** class. Whereas the color enumerated constants for **System.Drawing** colors are found in the **Color** class, in UWP they are found in the **Colors** class.

Example:

System.Drawing **UWP**

```
C#      Color c = Color.Red;          Color c = Colors.Red;
```

Color.FromArgb and Color.Rgb

Custom RGB colors were defined in .Net using the Color.FromArgb static method. It had a couple of overrides, one with four arguments, for a full ARGB color with alpha blending specification, and one with just three, for RGB. The UWP **Color** class has only one static methods used to define RGB colors: Color.FromArgb for the full ARGB specification.

System.Drawing

```
C#      Color c = Color.FromArgb(127, 255,0,0)
        Color c = Color.FromArgb(255,0,0)
```

UWP

```
C#      Color c = Color.FromArgb(127, 255,0,0)
```

Linestyle Constants

Line styles are used to specify whether a line is solid, dashed, dotted, or some combination of dot and dash styles. The line style class used in the .Net Forms based version of QCChart2D is derived from the **System.Drawing.DashStyle** class. Their is no UWP equivlaent DashStyle class. So we use our own constants defined in the ChartObj class.

Example:

System.Drawing

```
C#  DashStyle ls = DashStyle.Solid;
```

UWP

```
        public    const int LS_SOLID = 0;
        public    const int LS_DASH_8_4 = 1;
        public    const int LS_DASH_4_4 = 2;
        public    const int LS_DASH_4_2 = 3;
        public    const int LS_DASH_2_2 = 4;
        public    const int LS_DOT_1_1 = 5;
        public    const int LS_DOT_1_2 = 6;
        public    const int LS_DOT_1_4 = 7;
```

```
        public    const int LS_DOT_1_8 = 8;
        public    const int LS_DASH_DOT = 9;
```

```
C#  int ls = ChartObj.LS_SOLID;
```

Fonts

It is strange, but UWP does not encapsulate font properties into a single class, like the **System.Drawing.Font** class. Instead, UWP text objects which require a font specification use separate properties for the font name, font size, font style, and the font weight. We found this to be inconvenient, so we created a simple **ChartFont** class which superficially looks like the old **System.Drawing.Font** class. That way we could keep source codes consistent.

System.Drawing

C#

```
Font  theFont = new Font("Microsoft Sans Serif",10,FontStyle.Regular);
```

UWP

References to **Font** are replaced with the QCChart2D class **ChartFont**. Since the UWP **FontStyle** type does not include bold options, an additional constructor is added which has a **FontWeight** parameter. Specify the font weight using one of the **FontWeights** enumerated constants.

C#

```
ChartFont theFont = new ChartFont("Microsoft Sans Serif", 10,  FontStyle.Normal);
ChartFont theFont = new ChartFont("Microsoft Sans Serif", 10,  FontStyle.Normal,
FontWeights.Bold);
```

Font Style Constants

The **System.Drawing.FontStyle** enumerated constants are different than the UWP FontStyles constants.

.Net Forms	UWP
Bold	
Italic	Italic
	Oblique
Regular	Normal
Strikeout	
Underline	

Font Weight Constants

The UWP **FontWeights** enumerated constants include: Black, Bold, ExtraBlack, ExtraBold, ExtraLight, Light, Medium, Normal, SemiBold, SemiLight and Thin.

Grid

Grid is a panel class used everywhere in UWP programs for the layout of visual objects. Rather than force programmers to fully qualify the QCChart2D **Grid** class, which does something entirely different than the UWP **Grid** class, we changed the name of our grid to **ChartGrid**. Otherwise, the parameters remain the same as our original QCChart2D **Grid** class.

UWP

C#

```
ChartAttribute attrib3 = new ChartAttribute(Colors.Gray, 1, ChartObj.LS_DOT_1_4);
ChartGrid ygrid = new ChartGrid(xAxis, yAxis, ChartObj.Y_AXIS, ChartObj.GRID_MAJOR);
ygrid.SetChartObjAttributes(attrib3);
chartVu.AddChartObject(ygrid);
```

Mouse Button Constants

UWP events use different mouse button constants than .Net Forms based programming. The .Net Forms mouse event constants are found in S**ystem.Windows.Forms.MouseButtons**, while the UWP mouse constants are found in the QCChart2D class **MouseButton.**

```
public const int Left = 1;
public const int Right = 2;
public const int Middle = 4;
```

```
MoveObj mouselistener = new MoveObj(chartVu,MouseButton.Left);
```

Mouse Event Arguments

In .Net Forms programming, the MouseDown, MouseUp, and MouseMove events use the **System.Windows.Forms.MouseEventArgs** data type. In UWP programming, the use of the mouse identifier is depracated, and everything uses a generic Pointer term, since touch screens don't use a mouse. the MouseDown, MouseUp and MouseMove events use the **PointerRoutedEventArgs** type.

.Net Forms

C#

```
protected  void OnMouseDown( MouseEventArgs e)
 protected  void OnMouseUp( MouseEventArgs e)
```

```
protected  void OnMouseMove( MouseEventArgs e)
```

UWP

C#

```
protected  void OnMouseDown( PointerRoutedEventArgs e)
 protected  void OnMouseUp( PointerRoutedEventArgs e)
 protected  void OnMouseMove( PointerRoutedEventArgs e)
```

Timers

It your program does any real-time updates, it probably uses a timer class. The .Net Forms timer class is **System.Timers.Timer** while the UWP timer class is Windows.UI.Xaml**.DispatcherTimer**. They are similar in function with slightly different properties you must set.

.Net Forms

C#

```
System.Timers.Timer timer1 = new System.Timers.Timer();
timer1.Enabled = true;
timer1.Interval = 300;
timer1.Elapsed += new System.Timers.ElapsedEventHandler(timer1_Elapsed);
```

UWP

C#

```
DispatcherTimer timer1 =
    new DispatcherTimer();
timer1.Interval = TimeSpan.FromMilliseconds(2000);
timer1.Tick += new EventHandler<object>(this.timer1_Elapsed);
timer1.Start();
```

3. Chart Datasets

ChartDataset
- **SimpleDataset**
  - **TimeSimpleDataset**
  - **ElapsedTimeSimpleDataset**
  - **ContourDataset**
  - **EventSimpleDataset**
- **GroupDataset**
  - **TimeGroupDataset**
  - **ElapsedTimeGroupDataset**
  - **EventGroupDataset**

The dataset classes organize the numeric data associated with a plot object. Plot objects are chart objects derived from the **ChartPlot** class. There are two major types of data supported by the dataset classes. The first is simple xy data, where for every x-value there is one y-value. The second data type is group data, where every x-value can have one or more y-values. A couple of variants of the simple xy datasets include a simple dataset type that can substitute **ChartCalendar** values, or **TimeSpan** as the x- or y-values. Also, there is a dataset type that is used to plot contour data.

Except in the case of the ChartEvent datasets (EventSimpleDataset and EventGroupDatast), copies of the original data arrays are stored. The original source data can be deleted once the dataset is created. If you want to make any changes to the data, you must change the data in the dataset, not the original source data. The ChartEvent datasets are different. Because they contain an array of ChartEvent objects, and these objects can be quite large, a copy of the ChartEvent objects is NOT made. Instead, the dataset classes reference the ChartEvent objects passed into the constructor.

Datasets can be initialized using CSV (comma separated value) files. The CSV file is a common file structure that can share data between spreadsheets, databases and word processing programs. Datasets can also write CSV files, loadable into other programs.

If you need to plot data stored in a database, either save the data as a CSV file, or read the data into arrays. Once the data is in either format, inititialize a dataset using the appropriate class and constructor.

The **ChartDataset** class is the abstract base class for all of the dataset classes. It contains data common to all dataset classes, such as the x-value array, the number of x-values, the dataset name and the dataset type.

Loading data from a file

If you want to load data from a file, you must reference a .Net StorageFile you have access to.. Universal Windows Apps do not give you access to the entire file system. Instead you are limited to a few folders. The rules keep changing though, so in general assume that you need to get a valid StorageFolder for any folder you want to read data from. There are two main folders the program has Read access to: the Installation Folder of the software, and the Data Folder. Example of how to get a StorageFolder for both of these is seen below.

```
StorageFolder dataFolder = Windows.Storage.ApplicationData.Current.LocalFolder;

StorageFolder installationFolder =
Windows.ApplicationModel.Package.Current.InstalledLocation;
```

If you want to read a dataset, you can use either the *dataFolder*, or the *installationFolder*. If you want to write a dataset, you can ONLY use the *dataFolder*, because the *installationFolder* is Read Only.

You can then read a specific file, located in the referenced StorageFolder, using asynchronous I/O routines built in the dataset classes. The read dataset routine must be called with the **await** operator, which means that the method the read dataset routine is placed in must be marked with the async modifier. The example below shows how to read a TimeSimpleDataset from the installation storage folder.

```
async void InitializeChart(ChartView chartVu)
{

    TimeSimpleDataset DOWDataset = new TimeSimpleDataset();

    int result = await DOWDataset.ReadTimeSimpleDataset(installationFolder,
"DOWCloseData.txt");
    if (result != 0)
       return;
    '
    '
    '
```

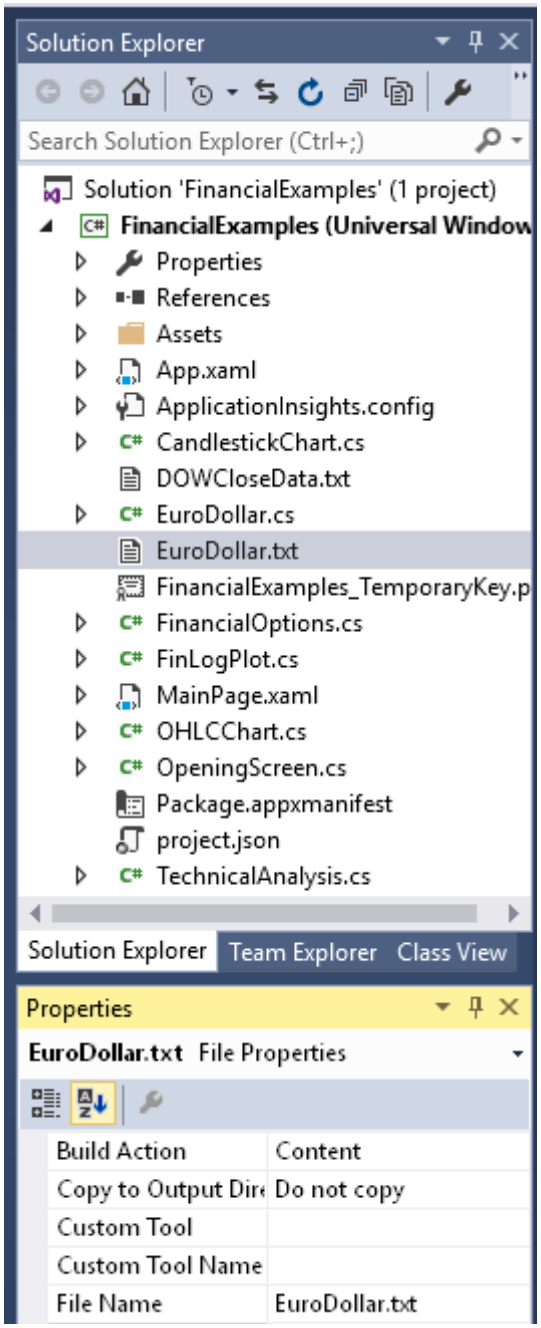
Important Note

It is very important that you load the data before the chart is created. Since the data is loaded using asynchronous routines, if you load them in one method (asyncronous), but plot the chart in another method (synchronous), it is quite possible to call the data load method before the chart drawing method, but have the chart drawing method complete before the data is actually loaded. So make sure that you have the synchronous and asynchronous parts of the program structured so that the data is present when the chart is defined, because the auto-axis routines, which rely on data, will fail.

In our example, the data is always loaded in the chart building method, using the **await** operator..So the method will not continue until the data is completely loaded. Then the method continues with the chart building code. That way the data is always present when the chart building code is called.

Important Note

The data files loaded in this fashion are stored in the projects main folder, next to the *.cs and *.xaml files.You must set the Build Action property of the file to Content. You do this by right clicking on the file in the VS Solution Explorer and selecting Properties. Then in the Properties table set the Build Action property to Content.



It is important to use the correct method for the given class.

Simple Datasets - which consist of one x-vector and one y-vector of values

Simple Dataset class	**Method for reading data**
SimpleDataset	ReadSimpleDataset
TimeSimpleDataset	ReadTimeSimpleDataset
ElapsedTimeSimpleDataset	ReadElapsedTimeSimpleDataset
EventSimpleDataset	ReadEventSimpleDataset

Group Datasets - which consist of one x-vector and one or more y-vectors of values

Group Dataset class	**Method for reading data**
GroupDataset	ReadGroupDataset
TimeGroupDataset	ReadTimeGroupDataset
ElapsedTimeGroupDataset	ReadElapsedTimeGroupDataset
EventGroupDataset	ReadEventGroupDataset
ContourDataset	ReadContourtDataset

Saving Data to a File

As previously mentioned, data can only be saved to the Data Storage File location, pointed to the by the code below:

```
StorageFolder dataFolder = Windows.Storage.ApplicationData.Current.LocalFolder;
```

This folder is read/write, so the program can write and read data to/from location. The basic code is very similar to that used for reading a file. You use the data file StorageFolder location, and call the appropriate I/O routine in the dataset class. You must call it using the await operator and the containing method must be marked with the async modifier. The example below shows how to write a TimeSimpleDataset to the data storage folder.

```
async void InitializeChart(ChartView chartVu)
{

    TimeSimpleDataset DOWDataset = new TimeSimpleDataset();
    int result = await DOWDataset.WriteTimeSimpleDataset(dataFolder, "DOWCloseData2.txt");

   if (result != 0)
       return;
    '
    '
    '
```

It is important to use the correct method for the given class.

Simple Datasets - which consist of one x-vector and one y-vector of values

Simple Dataset class	Method for writing data
SimpleDataset	WriteSimpleDataset
TimeSimpleDataset	WriteTimeSimpleDataset
ElapsedTimeSimpleDataset	WriteElapsedTimeSimpleDataset
EventSimpleDataset	WriteEventSimpleDataset

Group Datasets - which consist of one x-vector and one or more y-vectors of values

Group Dataset class	Method for writing data
GroupDataset	WriteGroupDataset
TimeGroupDataset	WriteTimeGroupDataset
ElapsedTimeGroupDataset	WriteElapsedTimeGroupDataset
EventGroupDataset	WriteEventGroupDataset
ContourDataset	WriteContourDataset

Simple Numeric Dataset

Class SimpleDataset

ChartObj
|
+--ChartDataset
|
+--SimpleDataset

The **SimpleDataset** class represents simple floating point xy data, where for every x-value there is one y-value. The number of xy data points in a simple dataset is referred to as the number of columns, or as the property NumberDatapoints Think of a spreadsheet file that looks like:

x-values	x[0]	x[1]	x[2]	x[3]	x[4]	x[5]
y-values	y[0]	y[1]	y[2]	y[3]	y[4]	y[5]

number of xy data pairs = NumberDatapoints = NumberColumns = 6

This would be the ROW_MAJOR format if the data were stored in a CSV file.

It has two main constructors. This constructor creates a dataset using the x- and y-values stored in arrays.

SimpleDataset constructors

```
[C#]
public SimpleDataset(
   string sname,
   double[] x,
   double[] y
);
```

sname	Specifies the name of the dataset.
x	An array that specifies the x-values of a dataset.
y	An array that specifies the y-values of a dataset. The length of the y array must match the length of the x array.

The number of data points is the value of x.Length property. The x and y arrays must be the same length and every element must be initialized to a valid value. All values in the arrays are plotted. If the data is outside of the current chart scale the values will be clipped.

You can retrieve references to the internal arrays used to store the data using the **SimpleDataset** methods **GetXData** and **GetYData**. Change the values in the data using these references. You can also modify a point at a time using one of the **SetDataPoint** methods. If you need to add new points to a dataset, increasing its size, use one of the **AddDataPoint**, or **InsertDataPoint** methods. Delete data points using the **DeleteDataPoint** method. In order to see the modified dataset, force the graph to redraw using **ChartView.UpdateDraw** method. The indexed accessor property of the **SimpleGroupDataset** will get or set a datapoint as a **Point2D** object.

Example of creating simple datasets from numeric arrays

[C#]

```
double [] x1 = {10, 20, 30, 40, 50};
double [] y1 = {9, -21, 20, 40, 30};
SimpleDataset Dataset1 = new SimpleDataset("First", x1, y1);

int n2  = 9;
double []x2 = new double[n2]; //  dimension is size, not upper limit
double []y2 = new double[n2]; //  dimension is size, not upper limit

x2[0] = 5;
x2[1] = 7;
.
.
x2[n2 - 1] = 100;
```

```
y2[0] = 15;
y2[1] = 25;
.
.
y2[n2 - 1] = 100;
SimpleDataset Dataset2 = new SimpleDataset("Second", x2, y2);
```

Example of modifying simple dataset elements using the indexed accessor property.

[C#]

```
// Define a simple dataset
SimpleDataset Dataset1 = new SimpleDataset("First",x1,y1);
Point2D datapoint = Dataset1[0]; // Get the xy point at index 0 in the dataset
if datapoint.X < 0 datapoint.X =  Math.Abs(datapoint.X); // arbitrary
Dataset1[0]  = datapoint; // Change the datapoint
```

Example of creating simple datasets from numeric arrays

[C#]

```
double [] x1 = {10, 20, 30, 40, 50};
double [] y1 = {9, -21, 20, 40, 30};
SimpleDataset Dataset1 = new SimpleDataset("First", x1, y1);

int n2  = 9;
double []x2 = new double[n2]; //  dimension is size, not upper limit
double []y2 = new double[n2]; //  dimension is size, not upper limit

x2[0] = 5;
x2[1] = 7;
.
.
x2[n2 - 1] = 100;

y2[0] = 15;
y2[1] = 25;
.
.
y2[n2 - 1] = 100;
SimpleDataset Dataset2 = new SimpleDataset("Second", x2, y2);
```

Read data from a file using the SimpleDataset ReadSimpleDataset method, passing in a StorageFolder object, and a file name. The method containing the WriteSimpleDataset calls must be market *async*.

Example of reading and writing a simple dataset from a CSV file

[C#]

```
StorageFolder dataFolder = Windows.Storage.ApplicationData.Current.LocalFolder;


int result = 0;
result = await Dataset1.WriteSimpleDataset(dataFolder,"Dataset11.csv");
result = await Dataset2.WriteSimpleDataset(dataFolder,"Dataset12.csv");
result = await Dataset3.WriteSimpleDataset(dataFolder,"Dataset13.csv");
.
.
.
// Read it back in just as a test
result = await Dataset1.ReadSimpleDataset(dataFolder,"Dataset11.csv");
result = await Dataset2.ReadSimpleDataset(dataFolder,"Dataset12.csv");
result = await Dataset3.ReadSimpleDataset(dataFolder,"Dataset13.csv");
```

Example of modifying simple dataset elements using the indexed accessor property.

[C#]

```
// Define a simple dataset
SimpleDataset Dataset1 = new SimpleDataset("First",x1,y1);
Point2D datapoint = Dataset1[0]; // Get the xy point at index 0 in the dataset
if datapoint.X < 0 datapoint.X =  Math.Abs(datapoint.X); // arbitrary
Dataset1[0]  = datapoint; // Change the datapoint
```

Simple Date/Time Dataset

Class TimeSimpleDataset

ChartObj
|
+--ChartDataset
|
+--SimpleDataset
|
+-TimeSimpleDataset

The **TimeSimpleDataset** uses **ChartCalendar** dates as one set of the x- or y-values, and floating point values as the other. *Starting with Revision 2.0 of the software*, you can have **ChartCalendar** dates as either the x- or y-values in a **TimeSimpleDatset**. **ChartCalendar** values are actually stored internally as their equivalent millisecond values. The **TimeSimpleDataset** class adds a large number of methods to the **SimpleDataset** class that make it easy to create and modify datasets that use **ChartCalendar** values.

Note - Do **not** use the **TimeSimpleDataset** if you want to display data using the elapsed time. The **TimeSimpleDataset** uses a full GregorianCalendar date/time and it is not suitable for the display of elapsed time, since time intervals do not have an explicit date, i.e. 10/11/2008. Use the **ElapsedTimeSimpleDataset** class, in combination with an **ElapsedTimeCoordinateSystem**, if you plan to create an elapsed time chart.

It has two main constructors. The following constructor creates a time dataset using the x- and y-values stored in arrays.

TimeSimpleDataset constructors

```
[C#]
public TimeSimpleDataset(
   string sname,
   ChartCalendar[] x,
   double[] y
);
public TimeSimpleDataset(
   string sname,
   double[] x
   ChartCalendar[] y,
);
```

sname	Specifies the name of the dataset.
x	An array that specifies the x-values (either *doubles* or ChartCalendar objects) of a dataset.
y	An array that specifies the y-values of a dataset. (either *doubles* or ChartCalendar objects). The length of the y array must match the length of the x array.

Either x- or y-values should be **ChartCalendar** based. The number of data points is the value of x.Length property. The x and y arrays must be the same length and every element must be initialized to a valid value. All values in the arrays are plotted. If the data is outside of the current chart scale the values will be clipped.

The next constructor creates a time dataset using the x- and y-values stored in a file that uses the CSV (Comma Separated Value) format. There are two ways to organize the numeric values in the data file. If you use the COLUMN_MAJOR format, the first column represents the time values and the second column the y-values. If you use the ROW_MAJOR format, the first row represents the time values and the second row the y-values. Use the **CSV.SetOrientation** method to initialize the csv argument for the proper data orientation.

```
[C#]
public TimeSimpleDataset(
   CSV csv,
   StorageFile sfile,
   int rowskip,
```

```
   int columnskip
);
```

csv	An instance of a **CSV** object.
sfile	A StorageFile object representing the input file
rowskip	Skip this many rows before starting the read operation.
columnskip	For each row of data, skip this many columns before starting this read operation.

A **DateTimeFormatInfo** object, and a date time format string, in the **CSV** class, control the interpretation of the **ChartCalendar** values. The format in the file must match the format specified for the **CSV** class. The underlying conversion mechanism calls the **DateTime.ToString(String formatstring, DateTimeFormatInfo info)** method for the conversion. The default format for the date time *formatstring* object is "M/dd/yy". Call the **SetDateTimeFomatString** method to change the default date time format. See the documentation for the .Net **DateTime.ToString** method to figure out the various formatting options for the date time format string. If you are into internationalization (and difficult to understand .Net documentation), you can also create your own **DateTimeFormatInfo** object**,** installing it in the **CSV** object using **CSV.SetTimeDateFormat** method**.** The date time format string and the **DateTimeFormatInfo** object apply to both CSV files used for input, and CSV files used for output. If an attempt is made to read date/time values that do not match the desired format, the data values are set to invalid date/time values.

You can retrieve a *copy* of the date time data using the **TimeSimpleDataset.GetTimeXData** (or **GetTimeYData)** method. It returns an array of **ChartCalendar** objects, and it is *not* a reference to the underlying data. The underlying data is stored as double values that represent the millisecond equivalent of the date time values. The **TimeSimpleDataset GetXData** and **GetYData** methods return references to the underlying numeric data**.** You can also modify a point at a time using one of the **SetTimeDataPoint, SetTimeXDataValue** (or **SetTimeYDataValue)** and **SetYDataValue** (or **SetXDataValue**) methods. If you need to add new points to the dataset, increasing its size, use one of the **AddTimeDataPoint**, or **InsertTimeDataPoint** methods. Delete data points using the **DeleteTimeDataPoint** method. In order to see the modified dataset, force the graph to redraw using **ChartView.UpdateDraw** method.

.

Example of creating a time simple datasets

[C#]

```
int nnumpnts = 32;
```

```
ChartCalendar [] x1= new ChartCalendar[nnumpnts];
double []y1 = new double[nnumpnts];
double []y2 = new double[nnumpnts];
y1[0] = 100;
y2[0] = 30;
x1[0] = (ChartCalendar) currentdate.Clone();
currentdate.Add(ChartObj.MONTH,3);
for (i=1; i < nnumpnts; i++)
{
    x1[i] = (ChartCalendar) currentdate.Clone();
    y1[i] += y1[i-1] +  (5 + i) * (0.75 - ChartSupport.GetRandomDouble());
    y2[i] +=  y2[i-1] +  (15 + i) * (0.95 - ChartSupport.GetRandomDouble());
    currentdate.Add(ChartObj.MONTH,3);
}
TimeSimpleDataset Dataset1 = new TimeSimpleDataset("Sales",x1,y1);
TimeSimpleDataset Dataset2 = new TimeSimpleDataset("Expenses",x1,y2);
```

Read data from a file using the TimeSimpleDataset ReadTimeSimpleDataset method, passing in a StorageFolder object, and a file name. The method containing the ReadTimeSimpleDataset or WriteTimeSimpleDataset calls must be market *async*.

Example of reading and writing a time simple dataset from a CSV file

[C#]

```
    StorageFolder dataFolder = Windows.Storage.ApplicationData.Current.LocalFolder;


    int result = 0;
    result = await Dataset1.WriteTimeSimpleDataset(dataFolder,"Dataset11.csv");
    result = await Dataset2.WriteTimeSimpleDataset (dataFolder,"Dataset12.csv");
    result = await Dataset3.WriteTimeSimpleDataset (dataFolder,"Dataset13.csv");
    .
    .
    .
    // Read it back in just as a test
    result = await Dataset1.ReadTimeSimpleDataset(dataFolder,"Dataset11.csv");
    result = await Dataset2.ReadTimeSimpleDataset(dataFolder,"Dataset12.csv");
    result = await Dataset3.ReadTimeSimpleDataset(dataFolder,"Dataset13.csv");
```

Simple Elapsed Time Dataset

Class ElapsedTimeSimpleDataset

ChartObj

|

+--ChartDataset

|

+--SimpleDataset

|

+-ElapsedTimeSimpleDataset

The **ElapsedTimeSimpleDataset** class uses **TimeSpan** values as one set of the x- or y-values, and floating point values as the other. **TimeSpan** values are actually stored internally as their equivalent millisecond values.

It has two main constructors. The following constructor creates a time dataset using the x- and y-values stored in arrays.

ElapsedTimeSimpleDataset constructors

```
[C#]
public ElapsedTimeSimpleDataset(
   string sname,
   TimeSpan [] x,
   double[] y
);
public ElapsedTimeSimpleDataset(
   string sname,
   double[] x
   TimeSpan [] y,
);
```

sname	Specifies the name of the dataset.
x	An array that specifies the x-values (either *doubles* or **TimeSpan** objects) of a dataset.
y	An array that specifies the y-values of a dataset. (either *doubles* or **TimeSpan** objects). The length of the y array must match the length of the x array.

Either x- or y-values should be **TimeSpan** based. The number of data points is the value of x.Length property. The x and y arrays must be the same length and every element must be initialized to a valid value. All values in the arrays are plotted. If the data is outside of the current chart scale the values will be clipped.

The next constructor creates an elapsed time dataset using the x- and y-values stored in a file that uses the CSV (Comma Separated Value) format. There are two ways to organize the numeric values in the data file. If you use the COLUMN_MAJOR format, the first column represents the time values and the second column the y-values. If you use the ROW_MAJOR format, the first row represents the time values and the second row the y-values. Use the **CSV.SetOrientation** method to initialize the csv argument for the proper data orientation.

```
[C#]
public ElapsedTimeSimpleDataset(
   CSV csv,
   StorageFile sfile,
   int rowskip,
   int columnskip
);
```

csv	An instance of a **CSV** object.
sfile	A StorageFile object representing the input file
rowskip	Skip this many rows before starting the read operation.
columnskip	For each row of data, skip this many columns before starting this read operation.

The only supported format for elapsed time values in a CSV file is d.hh.mm.ss.fff, (3.14:23:12.333 as an example of an elapsed time of three days, 14 hours, 23 minutes, 12 seconds and 333 milliseconds).

You can also modify a point at a time using **SetElapsedTimeXDataValue** (or **SetElapsedTimeYDataValue**) if you are using **TimeSpan** objects**,** and **SetYDataValue** or **SetXDataValue**) if you use millisecond values. If you need to add new points to the dataset, increasing its size, use one of the **AddDataPoint**, or **InsertDataPoint** methods. Delete data points using the **DeleteDataPoint** method. In order to see the modified dataset, force the graph to redraw using **ChartView.UpdateDraw** method.

.

Example of creating a simple elapsed time datasets, extracted from the NewDemosRev2.ElapsedTimeChart example program.

[C#]

```
int  numPoints = 100;
TimeSpan[] x1 = new TimeSpan[numPoints];
double []y1 = new double[numPoints];
double []y2 = new double[numPoints];
int i;
for (i=0; i < numPoints; i++)
{
    x1[i] =  TimeSpan.FromMilliseconds( i * 30 * 1000);  // 30000 milliseconds  increment
 // Or you can use seconds, and the FromSeconds method
//  x1[i] = TimeSpan.FromSeconds(i * 30);  // 30 seconds increment
     if (Math.Sin(x1[i].TotalSeconds / 20.0) > 0)
       y1[i] = 20.0 + 50.0 * (0.5 - ChartSupport.GetRandomDouble()) * (Math.Sin(-
x1[i].TotalSeconds / 5));
     else
      y1[i] = 20.0 + 5.0 * (0.5 - ChartSupport.GetRandomDouble()) * (Math.Sin(-
x1[i].TotalSeconds / 2));

   y2[i] = y1[i] + ((5 + 0.2 * x1[i].TotalSeconds) * (0.5 -
ChartSupport.GetRandomDouble()));
}
ElapsedTimeSimpleDataset Dataset1 = new ElapsedTimeSimpleDataset("First", x1, y1);
ElapsedTimeSimpleDataset Dataset2 = new ElapsedTimeSimpleDataset("Second", x1, y2);
```

Read data from a file using the ElapsedTimeSimpleDataset ReadElapsedTimeSimpleDataset method, passing in a StorageFolder object, and a file name. The method containing the ReadElapsedTimeSimpleDataset or WriteElapsedTimeSimpleDataset calls must be market *async*.

Example of reading and writing an elapsed time simple dataset from a CSV file

[C#]

```
StorageFolder dataFolder = Windows.Storage.ApplicationData.Current.LocalFolder;


int result = 0;
result = await Dataset1.WriteElapsedTimeSimpleDataset(dataFolder,"Dataset11.csv");
result = await Dataset2.WriteElapsedTimeSimpleDataset(dataFolder,"Dataset12.csv");
result = await Dataset3.WriteElapsedTimeSimpleDataset(dataFolder,"Dataset13.csv");
.
.
.
// Read it back in just as a test
result = await Dataset1.ReadElapsedTimeSimpleDataset(dataFolder,"Dataset11.csv");
result = await Dataset2.ReadElapsedTimeSimpleDataset(dataFolder,"Dataset12.csv");
result = await Dataset3.ReadElapsedTimeSimpleDataset(dataFolder,"Dataset13.csv");
```

Contour Plot Dataset

Class ContourDataset

ChartObj
|
+--ChartDataset
|
+--SimpleDataset
|
+-ContourDataset

The **ContourDataset** adds a third dimension (z-values) to the x- and y- values of the simple dataset. It is use exclusively with the contour plotting class, **ContourPlot**.

This constructor creates a new **ContourDataset** object that represents a surface formed by a regular grid in the xy plane. The number of objects in the **Point3D** array must equal (rows * columns) and must form an even grid in the xy plane.

ContourDataset constructors

```
[C#]
public ContourDataset(
```

```
   string sname,
   Point3D[] grid,
   int rows,
   int columns
);
```

This constructor creates a new **ContourDataset** object that represents a surface, not necessarily a regular grid. A triangulation algorithm calculates the interconnection of the vertices defining the surface.

```
[C#]
public ContourDataset(
   string sname,
   Point3D[] grid
);
```

This constructor creates a new **ContourDataset** object that represents a surface, not necessarily a regular grid. A triangulation algorithm calculates the interconnection of the vertices defining the surface. The length of the x, y and z arrays must match.

```
[C#]
public ContourDataset(
   string sname,
   double[] x,
   double[] y,
   double[] z
);
```

This constructor creates a new **ContourDataset** object defined using the supplied **SurfaceFunction** class, evaluated for the range x1,y1 to x2,y2 at intervals equal to (x2-x1)/columns for the x direction, and (y2-y1)/rows in the y direction. This forms a regular grid surface.

```
[C#]
public ContourDataset(
   string sname,
   int rows,
   int columns,
   double x1,
   double y1,
   double x2,
   double y2,
   SurfaceFunction sf
);
```

The next constructor creates a dataset using the x-, y- and z-values stored in a file that uses the CSV (Comma Separated Value) format. There are two ways to organize the numeric values in the data file. If you use the COLUMN_MAJOR format, the first column represents the x-values and the second and third columns the y- and z-values. If you use the ROW_MAJOR format, the first row represents the x-values and the second and third row the y- and z-values. Use the **CSV.SetOrientation** method to initialize the csv argument for the proper data orientation.

```
[C#]
public ContourDataset(
   CSV csv,
   StorageFile sfile,
```

```
   int rowskip,
   int columnskip
);
```

sname	Specifies the name of the dataset.
grid	An array, size [npoints] (or size [rows * columns]) of **Point3D** points, that specifies the xyz values of a dataset. Some of the constructors require the data points form a regular grid in the xy plane. A regular grid is one where the x-increment between adjacent x-values is fixed, as is the y-increment. The x-increment and the y-increment do not have to be the same.
rows	Specifies the number of rows (in the y direction) in the regular grid. Also specifies the number of rows (or y-values) to evaluate the function over in the constructor that uses a **SurfaceFunction** argument.
columns	Specifies the number of columns (in the x direction) in the regular grid. Also specifies the number of columns (or y-values) to evaluate the function over in the constructor that uses a **SurfaceFunction** argument.
npoints	Specifies the number of xyz data point triplets in the grid array.
x	An array, size [npoints] of double that specifies the x-values of the dataset. The length of the y and z arrays must equal x.Length.
y	An array, size [npoints] of double that specifies the y-values of the dataset.
z	An array, size [npoints] of double that specifies the z-values of the dataset.
x1, y1, x2, y2	The **SurfaceFunction** sf is evaluated for the range x1,y1 to x2, y2.
sf	The dataset data points are created by evaluating the **SurfaceFunction** across the range x1,y1 to x2, y2.
csv	An instance of a **CSV** object.
sfile	A StorageFile object representing the input file
rowskip	Skip this many rows before starting the read operation.
columnskip	For each row of data, skip this many columns before reading the first value from the row.

Example of creating a contour dataset from an array of Point3D

[C#]

```
int nrows=11, ncols=11;
int i, j, count=0;
double x, y, z;
double startx = -6.0, starty = -6.0;
double stepx = 12.0/(nrows-1), stepy = 12.0/(ncols-1);
Point3D []pointarray;

pointarray = new Point3D[nrows * ncols];
x = startx;
y = starty;
for (i = 0; i < nrows; i++)
{
   x = startx;
   for (j=0; j < ncols; j++)
   {
     pointarray[count] = new Point3D();
     z = 2000 + ( 950 * Math.Sin(Math.Sqrt(x*x+ y*y)));
     pointarray[count].SetLocation(x, y, z);
     x += stepx;
     count++;
    }
    y += stepy;
}
// This method triangulates data into a surface
ContourDataset dataset1 =  new ContourDataset("Contour Dataset",pointarray);
// This method uses the characteristic that the data is an even spaced grid.
ContourDataset dataset2=
    new ContourDataset("Contour Dataset",pointarray, nrows, ncols);
```

Example of creating a contour dataset from a function

[C#]

```
ContourDataset dataset1 = null;
class ZValueFunctionClass: SurfaceFunction
{
    public override double CalcZValue(double x, double y)
    {
        double z;
        x = x + 1.75 * (ChartSupport.GetRandomDouble() - 0.5);
        y = y + 1.75 * (ChartSupport.GetRandomDouble() - 0.5);
        z = 1500 + (1500.0 * Math.Sin(Math.Sqrt(x*x+ y*y)));
        return z;
    }
}

void CreateRegularGridPolysurface()
{
    ZValueFunctionClass zValueFunction = new ZValueFunctionClass();
    dataset1 =   new ContourDataset("Contour ChartDataset",11, 11,
                      -6.0, -6.0, 6.0, 6.0, zValueFunction);
}
```

Read data from a file using the ContourDataset ReadContourDataset method, passing in a StorageFolder object, and a file name. The method containing the ReadContourDataset or WriteContourDataset calls must be market *async*.

Example of reading and writing a contour dataset from a CSV file

[C#]

```
StorageFolder dataFolder = Windows.Storage.ApplicationData.Current.LocalFolder;


int result = 0;
result = await Dataset1.WriteContourDataset(dataFolder,"Dataset11.csv");
result = await Dataset2.WriteContourDataset (dataFolder,"Dataset12.csv");
result = await Dataset3.WriteContourDataset (dataFolder,"Dataset13.csv");
.
.
.
// Read it back in just as a test
result = await Dataset1.ReadContourDataset(dataFolder,"Dataset11.csv");
result = await Dataset2.ReadContourDataset (dataFolder,"Dataset12.csv");
result = await Dataset3.ReadContourDataset (dataFolder,"Dataset13.csv");
```

Simple Event Dataset

Class

EventSimpleDataset

ChartObj
|
+--ChartDataset
|
+--SimpleDataset
|
+-EventSimpleDataset

Background for ChartEvent datasets

Most coordinate systems used in plotting represent a continuous domain. In the QCChart2D software, these includes linear, logarithmic, simple time/date, elapsed time, polar, and antenna coordinate systems. The one exception is a variant of the time/date scale which allows for periodic, yet discontinuous time. In the time/date scale case, it is possible to remove weekends from the time scale, and to define the hours of the day to be some subset of the standard 24-hour cycle. The most often used example is the stock trading day used in the US, which is from 9:30 to 16:00, and does not include weekends.

Unfortunately, the time coordinate system, even with discontinuous time, is still insufficient to plot the great variety of time plots needed in the financial services, and other, industries. Some of these special requirements are:

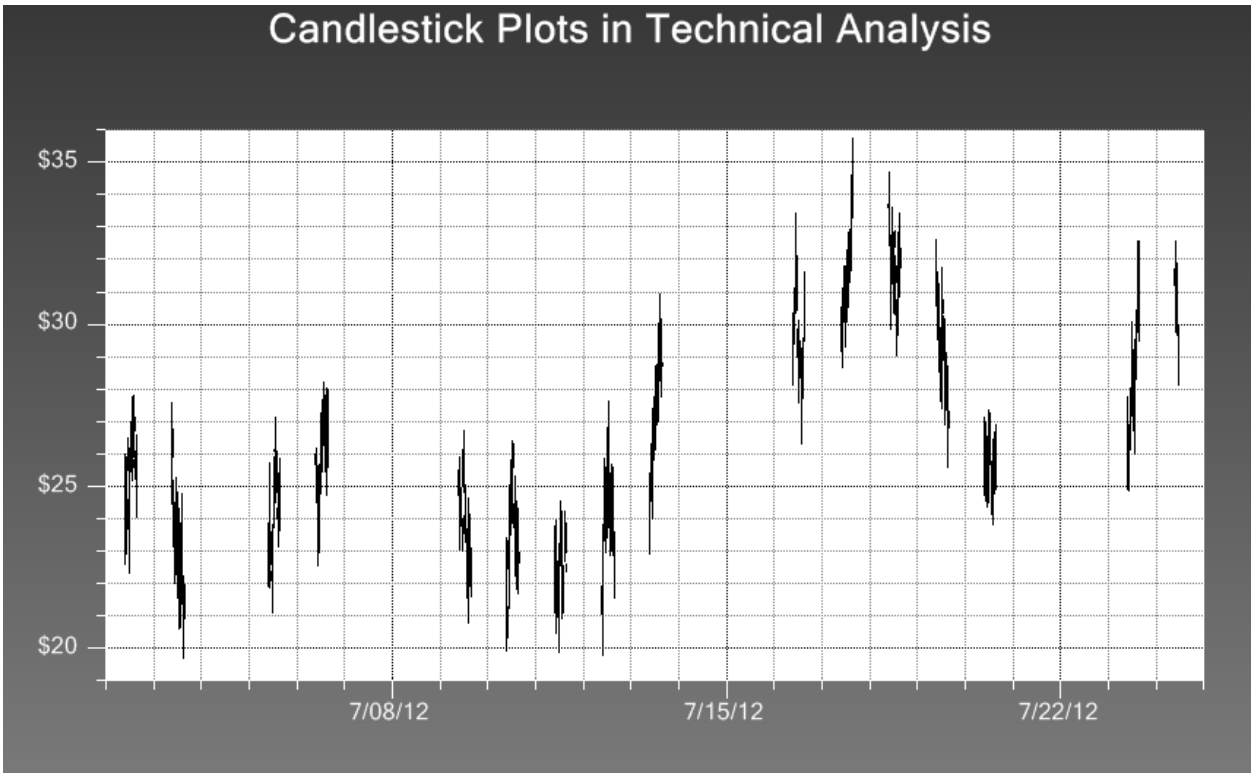
- Must be able to remove arbitrary (non-periodic) days, holidays for example, from the time/date scale.
- Allow for a sub range of a day which crosses 24:00, i.e. 18:00 to 3:00.

- Allow for multiple, active time ranges within the same 24-hour period, i.e. 9:00 AM to 12:00 and 14:00 to 18:00.
- Smooth panning and zooming of data across discontinuous time boundaries.
- Allow for exceptions to the predefined set of rules. For example, be able to include a weekend day, or a specific set of hours normally excluded from the scale.

These requirements are common enough that we wanted to address them with new coordinate system, dataset and axis classes, which can accommodate any set of continuous, or discontinuous time/date values, but not waste display space on gaps where no data exists. Rather than extend the existing time/date coordinate system (TimeCoordinates), we chose to create new coordinate system, EventCoordinates, which uses discrete events, rather than a continuous domain, as the basis for plotting data. The basis of the EventCoordinates system are the event dataset classes: EventSimpleDataset, and EventGroupDataset, and the underlying array of ChartEvent objects . Rather than define a plot using arrays of x- and y-values, a ChartEvent represents a specific point in time. The point in time has the following major properties as distinguishing elements:

- **Description** – A description of the event.
- **ShortDescription** - An abbreviated description of the event.
- **AxisLabel** – A string which can be uses as an axis label for the event.
- **ToolTip** – A custom tool-tip which can be displayed if the event is clicked on.
- **Position** – The position of the event with respect to the underlying linear coordinate system.
- **TimeStamp** – The time stamp of the event. Indirectly related to the Position of the event in the coordinate system
- **NumericTimeStamp** – The numeric value of the time stamp in milliseconds + an offset (numericTimeStampOffset).
- **NumericValues** - One or more numeric y-values (a simple plot uses a single y-value, while a group plot uses an array of y-values.

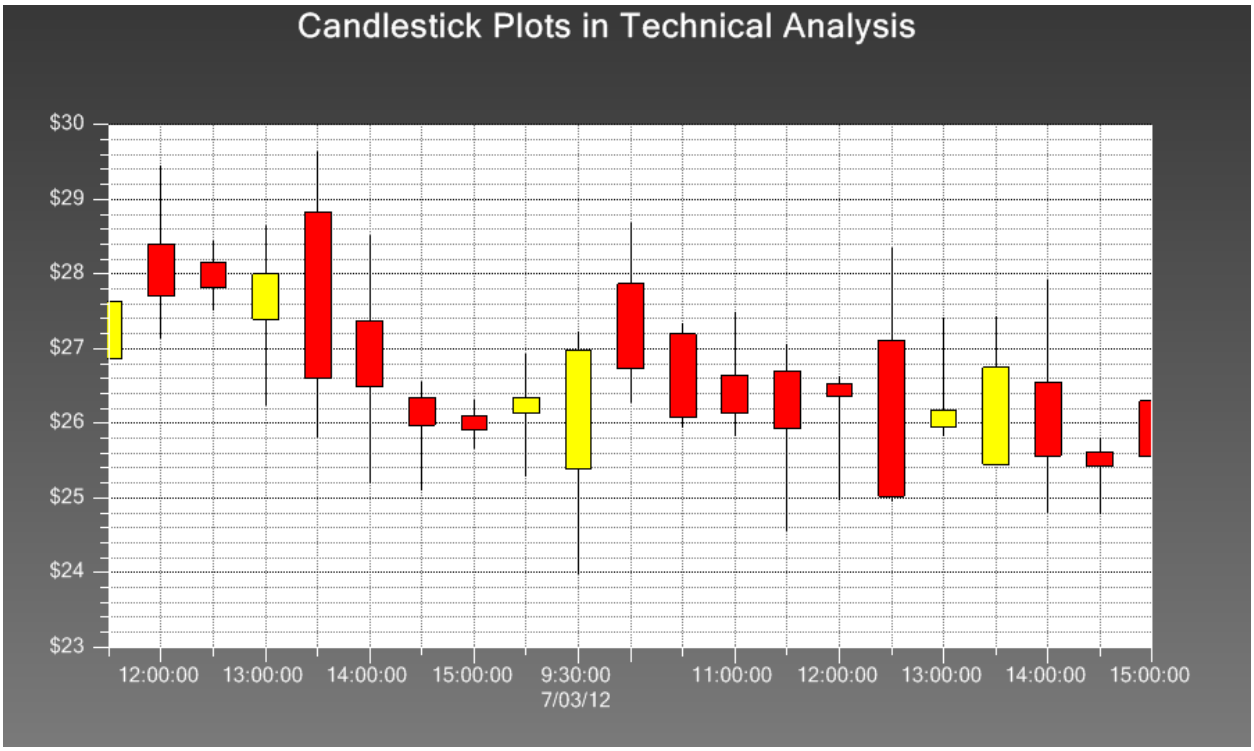
The ChartEvent class incorporates two x-value positioning properties, the Position and the TimeStamp, and one or more numeric y-values for each event. A single event therefore defines both the x-and y-values of the event in the underlying coordinate system. A collection, or array, of ChartEvent objects define the data for a plot, the same way as arrays of x- and y-values define a plot when using a simple dataset class with a Cartesian coordinate system. The critical element of the ChartEvent which permit it to be used for the plotting of discontinuous data is that the Position of the event in a chart is related, but, independent of the TimeStamp of the event. Event data can be positioned contiguously, and evenly spaced, in a chart, even if the time stamps of the events are not contiguous, or evenly spaced. Here is a simple example of a standard financial candlestick plot chart, using our TimeCoordinates class as the coordinate system, where the time/date data is not evenly spaced, and contains large gaps corresponding to weekends, and inactive hours of the day. The July 4th holiday is included in the range, and there is no data for that time interval either.



Contrast this to the similar data, using the same time range, plotted using the EventCoordinates class. Note how every event is evenly spaced with its neighbor. Gaps do not exist, since weekends, holidays, and unused hours are bridged over as if they do not exist.he same would be true for gaps due to holidays, and a varying number of work hours in a day.

Zooming in further, you can see the smooth transition across the July 4th holiday, and the following weekend.

Zoom in again, and you can see the smooth transition from one day to the next, even though the working hours are only a 9:30 to 16:00 subset of the 24 hours of a day.



This is accomplished because of the dual positioning values, Position and TimeStamp, of the ChartEvent class. Each element of a plot object (one of the candlestick objects in the plot above) is positioned in a simple linear coordinate system, starting at 0 and incrementing by 1 for each ChartEvent object. In the previous example, the first ChartEvent object is has a Position value of 0.0, and the last ChartEvent object has a position of 199, because there are 200 data points in the chart. This is what keeps the individual elements of a plot o bject evenly spaced, because the plot elements are positioned in the chart using the Position value, not the TimeStamp value. But, the associated x-axis (EventAxis) and x-axis labels objects (EventAxisLabels) look to the TimeStamp property for their values, not the Position property. What you end up with is the clean, evenly spaced look of a simple linear chart, with the axis tick marks and axis labeling of a dedicated time/date axis. The graph can be made to scroll (or pan) left to right, or re-scale along the y-axis, smoothly.

The EventSimpleDataset class is used to supply the simple plot classes: SimpleLinePlot, SimpleBarPlot, SimpleScatterPlot, and SimpleLineMarkerPlot, with data.

A ChartEvent can have one ore more y-values. In the case of an EventSimpleDataset, usually it will have a single value, as seen in the programming example below, one y-value for each x-value. In the EventGroupDataset, each ChartEvent object can have multiple y-values for each x-value.

EventSimpleDataset constructors

```
[C#]
public EventSimpleDataset(
   string sname,
```

```
   ChartEvent_[] ev
);
```

sname Specifies the name of the dataset.

ev An array of ChartEvent objects.

Create an array of ChartEvent objects, specifying the time-stamp and y-value for each ChartEvent object and use that to initialize a EventSimpleDataset.

The next constructor creates an EventSimpleDataset using the x- and y-values stored in a file that uses the CSV (Comma Separated Value) format. Only the COLUMN_MAJOR format is supported, where each row presents a ChartEvent object, and the columns are organized as: description, short description, x-axis string label, tool tip string, position, time stamp, numeric time stamp, y-value index (index for the y-value to use when the ChartEvent contains multiple y-values), and the y-values.

```
[C#]
public EventSimpleDataset(
   CSV csv,
   StorageFile sfile,
   int rowskip,
   int columnskip
);
```

csv An instance of a **CSV** object.

sfile A StorageFile object representing the input file

rowskip Skip this many rows before starting the read operation.

columnskip For each row of data, skip this many columns before starting this read operation.

A **DateTimeFormatInfo** object, and a date time format string, in the **CSV** class, control the interpretation of the **ChartCalendar** values. The format in the file must match the format specified for the **CSV** class. The underlying conversion mechanism calls the **DateTime.ToString(String formatstring, DateTimeFormatInfo info)** method for the conversion. The default format for the date time *formatstring* object is "M/dd/yy". Call the **SetDateTimeFomatString** method to change the default date time format. See the documentation for the .Net **DateTime.ToString** method to figure out the various formatting options for the date time format string. If you are into internationalization (and difficult to understand .Net documentation), you can also create your own **DateTimeFormatInfo** object**,** installing it in the **CSV** object using **CSV.SetTimeDateFormat** method**.** The date time format string and the **DateTimeFormatInfo** object apply to both CSV files used for input, and CSV

files used for output. If an attempt is made to read date/time values that do not match the desired format, the data values are set to invalid date/time values.

You can also modify a point at a time using **SetEvent.** If you need to add new points to the dataset, increasing its size, use one of the **AddEvent**, or **InsertEvent** methods. Delete data points using the **DeleteEvent** method. In order to see the modified dataset, force the graph to redraw using **ChartView.UpdateDraw** method.

.

Example of creating a simple event datasets, extracted from the ChartEventExamples.SimpleEventChart example program.

[C#]

```
int nnumpnts = 10;
ChartCalendar[] x1 = new ChartCalendar[nnumpnts];
double[] y1 = new double[nnumpnts];
ChartCalendar currentdate = new ChartCalendar(1998, ChartObj.JANUARY, 1);
ChartEvent[] chartevents = new ChartEvent[nnumpnts];

int i;

double startx = 1;
for (i = 0; i < nnumpnts; i++)
{
  if (i == 0)
  {
     y1[0] = 100;
     x1[0] = (ChartCalendar)currentdate.Clone();
     chartevents[0] = new ChartEvent(x1[0], startx, y1[0]);
     chartevents[0].AxisLabel = "XY" + "0";
     currentdate.Add(ChartObj.MONTH, 12);
  }
  else
  {
     x1[i] = (ChartCalendar)currentdate.Clone();
     y1[i] += y1[i - 1] + 25 * (0.55 - ChartSupport.GetRandomDouble());
     chartevents[i] = new ChartEvent(x1[i], i + startx, y1[i]);
     chartevents[i].AxisLabel = "XY" + i.ToString();
     currentdate.Add(ChartObj.MONTH, 12);
   }
 }

 theFont = new ChartFont("Microsoft Sans Serif", 10, FontStyle.Normal);

 EventSimpleDataset Dataset1 = new EventSimpleDataset("Actual Sales", chartevents);
 EventCoordinates pTransform1 = new EventCoordinates(Dataset1);
```

Read data from a file using the EventSimpleDataset ReadEventSimpleDataset method, passing in a StorageFolder object, and a file name. The method containing the ReadEventSimpleDataset or WriteEventSimpleDataset calls must be market async.

Example of reading and writing an event simple dataset from a CSV file

[C#]

```
StorageFolder dataFolder = Windows.Storage.ApplicationData.Current.LocalFolder;


int result = 0;
result = await Dataset1.WriteEventSimpleDataset(dataFolder,"Dataset11.csv");
result = await Dataset2.WriteEventSimpleDataset(dataFolder,"Dataset12.csv");
result = await Dataset3.WriteEventSimpleDataset(dataFolder,"Dataset13.csv");
.
.
.
// Read it back in just as a test
result = await Dataset1.ReadEventSimpleDataset(dataFolder,"Dataset11.csv");
result = await Dataset2.ReadEventSimpleDataset(dataFolder,"Dataset12.csv");
result = await Dataset3.ReadEventSimpleDataset(dataFolder,"Dataset13.csv");
```

Numeric Group Dataset

Class GroupDataset

ChartObj
|
+--ChartDataset
|
+--GroupDataset

The **GroupDataset** class represents group data, where every x-value can have one or more y-values. The number of x-values in a group plot is referred to as the number of columns or as **NumberDatapoints** and the number of y-values *for each x-value* is referred to as the number of rows, or **NumberGroups**. Think of spreadsheet file that looks like

x-values	x[0]	x[1]	x[2]	x[3]	x[4]	x[5]
y-values group #0	y[0,0]	y[0,1]	y[0,2]	y[0,3]	y[0,4]	y[0,5]
y-values group #1	y[1,0]	y[1,1]	y[1,2]	y[1,3]	y[1,4]	y[1,5]
y-values group #2	y[2,0]	y[2,1]	y[2,2]	y[2,3]	y[2,4]	y[2,5]

number of x-values = **NumberDatapoints** = NumberColumns = 6

number of y-values for each x-value = **NumberGroups** = NumberRows = 3

This would be the ROW_MAJOR format if the data were stored in a CSV file.

GroupDataset constructors

```
[C#]
public GroupDataset(
   string sname,
   double[] x,
   double[,] y
);
```

sname	Specifies the name of the dataset.
x	An array that specifies the x-values of a group dataset. The length of the x array sets the number of columns for the group dataset.
y	An array that specifies the y-values of a group dataset where y has the dimensions [number of rows, number of columns]. The number of rows in the y-array sets the number of groups in the group dataset. The number of columns in the y-array must match the length of the x-array.

The number of columns in the group dataset is the value of x.Length property. The number of columns in the y array must match the length of the x array and every element must be initialized to a valid value. All values in the arrays are plotted. If the data is outside of the current chart scale the values will be clipped.

The next constructor creates a dataset using the x- and y-values stored in a file that uses the CSV (Comma Separated Value) format. There are two ways to organize the numeric values in the data file. If you use the COLUMN_MAJOR format, the first column represents the x-values and subsequent columns represent the y-values, where each column is a group. If you use the ROW_MAJOR format, the first row represents the x-values and subsequent rows represent the y-values, where each row is a group. Use the **CSV.SetOrientation** method to initialize the csv argument for the proper data orientation.

```
[C#]
public GroupDataset(
   CSV csv,
   StorageFile sfile,
   int rowskip,
   int columnskip
);
```

csv	An instance of a **CSV** object.
sfile	A StorageFile object representing the input file
rowskip	Skip this many rows before starting the read operation.
columnskip	For each row of data, skip this many columns before reading the first value from the row.

You can retrieve references to the internal arrays used to store the data using the **GroupDataset** methods **GetXData** and **GetGroupData**. Change the values in the data arrays using these references. You can also modify a point at a time using one of the **SetYDataValue** and **SetXDataValue** methods. If you need to add new points to dataset, increasing its size, use one of the **AddGroupDataPoints**, or **InsertGroupDataPoints** methods. Delete data points using the **DeleteDataPoint** method. In order to see the modified dataset, force the graph to redraw using **ChartView.UpdateDraw** method.

Example of creating a group datasets from numeric arrays

[C#]

```
double []x1=  {10,20,30,40,50};
double [,]y1 = {{ 9,-21, 20,40,30},
                { 55,15,35,10,56},
                {15,25,15,30,40}};
GroupDataset Dataset11 = new GroupDataset("First",x1, y1);
```

Read data from a file using the GroupDataset ReadGroupDataset method, passing in a StorageFolder object, and a file name. The method containing the ReadGroupDataset or WriteGroupDataset calls must be market *async*.

Example of reading and writing a group dataset from a CSV file

[C#]

```
    StorageFolder dataFolder = Windows.Storage.ApplicationData.Current.LocalFolder;


    int result = 0;
    result = await Dataset1.WriteGroupDataset(dataFolder,"Dataset11.csv");
    result = await Dataset2.WriteGroupDataset(dataFolder,"Dataset12.csv");
    result = await Dataset3.WriteGroupDataset(dataFolder,"Dataset13.csv");
    .
    .
    .
    // Read it back in just as a test
    result = await Dataset1.ReadGroupDataset(dataFolder,"Dataset11.csv");
    result = await Dataset2.ReadGroupDataset(dataFolder,"Dataset12.csv");
    result = await Dataset3.ReadGroupDataset(dataFolder,"Dataset13.csv");
```

Date/Time Group Dataset

Class TimeGroupDataset

ChartObj

|

```
+--ChartDataset
       |
       +--GroupDataset
              |
              +-TimeGroupDataset
```

The **TimeGroupDataset** uses **ChartCalendar** dates as the x-values, and floating point numbers as the y-values. **ChartCalendar** values are actually stored internally as their equivalent millisecond values. The **TimeGroupDataset** class adds a large number of methods to the **GroupDataset** class that make it easy to create and modify datasets that use **ChartCalendar** values.

Note - Do **not** use the **TimeGroupDataset** if you want to display data using the elapsed time. The **TimeGroupDataset** uses a full GregorianCalendar date/time and it is not suitable for the display of elapsed time, since time intervals do not have an explicit date, i.e. 10/11/2008. Use the **ElapsedTimeGroupDataset** class, in combination with an **ElapsedTimeCoordinateSystem**, if you plan to create an elapsed time chart.

This constructor creates a new group **TimeGroupDataset** object where the x-values are **ChartCalendar** values and the y-values are floating point numbers.

TimeGroupDataset constructors

```
[C#]
public TimeGroupDataset(
   string sname,
   ChartCalendar[] x,
   double[,] y
);
```

sname	Specifies the name of the dataset.
x	An array of **ChartCalendar** dates, that specifies the x-values of a dataset. The length of the x array sets the number of columns for the group dataset.
y	An array that specifies the y-values of a group dataset where y has the dimensions [number of rows, number of columns]. The number of rows in the y-array sets the number of groups in the group dataset. The number of columns in the y-array must match the length of the x-array.

The number of columns in the group dataset is the value of x.Length property. The number of columns in the y array must match the length of the x array and every element must be initialized to a valid value. All values in the arrays are plotted. If the data is outside of the current chart scale the values will be clipped.

```
[C#]
public TimeGroupDataset(
   CSV csv,
```

```
   StorageFile sfile,
   int rowskip,
   int columnskip
);
```

csv	An instance of a **CSV** object.
sfile	A StorageFile object representing the input file
rowskip	Skip this many rows before starting the read operation.
columnskip	For each row of data, skip this many columns before reading the

There are two ways to organize the numeric values in the data file. If you use the COLUMN_MAJOR format, the first column represents the time values and subsequent columns represent the y-values, where each column is a group. If you use the ROW_MAJOR format, the first row represents the time values and subsequent rows represent the y-values, where each row is a group. Use the **CSV.SetOrientation** method to initialize the csv argument for the proper data orientation.

A **DateTimeFormatInfo** object, and a date time format string, in the **CSV** class, control the interpretation of the **ChartCalendar** values. The format in the file must match the format specified for the **CSV** class. The underlying conversion mechanism calls the **DateTime.ToString(String formatstring, DateTimeFormatInfo info)** method for the conversion. The default format for the date time *formatstring* object is "M/dd/yy". Call the **SetDateTimeFomatString** method to change the default date time format. See the documentation for the .Net **DateTime.ToString** method to figure out the various formatting options for the date time format string. If you are into internationalization (and difficult to understand .Net documentation), you can also create your own **DateTimeFormatInfo** object**,** installing it in the **CSV** object using **CSV.SetTimeDateFormat** method**.** The date time format string and the **DateTimeFormatInfo** object apply to both CSV files used for input, and CSV files used for output. If an attempt is made to read date/time values that do not match the desired format, the data values are set to invalid date/time values.

You can retrieve a *copy* of the date time data using the **TimeGroupDataset.GetTimeXData** method. It returns an array of **ChartCalendar** objects, and it is *not* a reference to the underlying data. The underlying data is stored as double values that represent the millisecond equivalent of the date time values. The **TimeGroupDataset GetXData** and **GetYData** methods return references to the underlying data**.** You can also modify a point at a time using one of the **TimeGroupDataset, SetTimeXDataValue** and **SetYDataValue** methods. If you need to add new points to dataset, increasing its size, use one of the **AddTimeGroupDataPoints**, or **InsertTimeGroupDataPoints** methods. Delete data points using the **DeleteDataPoint** method. In order to see the modified dataset, force the graph to redraw using **ChartView.UpdateDraw** method.

Example of creating a group time datasets

[C#]

```
int nNumPnts = 50, nNumGroups = 4;
int weekmode = ChartObj.WEEK_5D;
ChartCalendar []xValues=  new ChartCalendar[nNumPnts];
double [,]stockPriceData = new double[nNumGroups,nNumPnts];
double minval=0.0, maxval = 0.0;
int i;

ChartCalendar currentdate = new ChartCalendar();
ChartCalendar.SetTOD(currentdate,0,0,1);
currentdate = ChartCalendar.CalendarDaysAdd(currentdate, 1, weekmode);
    // Make sure not to start on a weekend
xValues[0] = (ChartCalendar) currentdate.Clone();
currentdate = ChartCalendar.CalendarDaysAdd(currentdate, 1, weekmode);
stockPriceData[3,0]  = 25;  // close
stockPriceData[0,0] = 25;   // open
stockPriceData[1,0] = 26;   // high
stockPriceData[2,0] = 24;   // low

for (i=1; i < nNumPnts; i++)
{
    xValues[i] = (ChartCalendar) currentdate.Clone();
    stockPriceData[3,i] += stockPriceData[3,i-1] +
      3 * (0.52 -   ChartSupport.GetRandomDouble()); // close
   stockPriceData[0,i] += stockPriceData[3,i] +
      2 * (0.5 - ChartSupport.GetRandomDouble()); // open
  minval = Math.Min(stockPriceData[3,i], stockPriceData[0,i]);
  maxval = Math.Max(stockPriceData[3,i], stockPriceData[0,i]);
  stockPriceData[1,i] = maxval + 1.5 * ChartSupport.GetRandomDouble();  // high
  stockPriceData[2,i] = minval - 1.5 * ChartSupport.GetRandomDouble();  // low
  currentdate = ChartCalendar.CalendarDaysAdd(currentdate, 1, weekmode);
}
TimeGroupDataset Dataset1 = new
       TimeGroupDataset("Stock Data",xValues,stockPriceData);
TimeGroupDataset Dataset1 = new
       TimeGroupDataset("Stock Data",xValues,stockPriceData);
```

Read data from a file using the TimeGroupDataset ReadTimeGroupDataset method, passing in a StorageFolder object, and a file name. The method containing the ReadTimeGroupDataset or WriteTimeGroupDataset calls must be market *async*.

Example of reading and writing a time group dataset from a CSV file

[C#]

```
    StorageFolder dataFolder = Windows.Storage.ApplicationData.Current.LocalFolder;


    int result = 0;
    result = await Dataset1.WriteTimeGroupDataset(dataFolder,"Dataset11.csv");
    result = await Dataset2.WriteTimeGroupDataset(dataFolder,"Dataset12.csv");
    result = await Dataset3.WriteTimeGroupDataset(dataFolder,"Dataset13.csv");
    .
    .
```

```
.
// Read it back in just as a test
result = await Dataset1.ReadTimeGroupDataset(dataFolder,"Dataset11.csv");
result = await Dataset2.ReadTimeGroupDataset(dataFolder,"Dataset12.csv");
result = await Dataset3.ReadTimeGroupDataset(dataFolder,"Dataset13.csv");
```

Elapsed Time Dataset

Class ElapsedTimeGroupDataset

ChartObj
|
+--ChartDataset
|
+--GroupDataset
|
+--ElapsedTimeGroupDataset

The **ElapsedTimeGroupDataset** class represents group data, where every x-value can have one or more y-values. The number of x-values in a group plot is referred to as the number of columns or as **NumberDatapoints** and the number of y-values *for each x-value* is referred to as the number of rows, or **NumberGroups**.

ElapsedTimeGroupDataset constructors

```
[C#]
public ElapsedTimeGroupDataset(
   string sname,
   TimeSpan[] x,
   double[,] y
);

public ElapsedTimeGroupDataset(
   string sname,
   double [] x,
   double[,] y
);
```

sname	Specifies the name of the dataset.
x	An array that specifies the x-values of a group dataset. The length of the x array sets the number of columns for the group dataset.
y	An array that specifies the y-values of a group dataset where y has the dimensions [number of rows, number of columns]. The number of rows in the y-array sets the number of groups in the group dataset. The number of columns in the y-array must match the length of the x-array.

The number of columns in the group dataset is the value of x.Length property. The number of columns in the y array must match the length of the x array and every element must be initialized to a valid value. All values in the arrays are plotted. If the data is outside of the current chart scale the values will be clipped.

The next constructor creates a dataset using the x- and y-values stored in a file that uses the CSV (Comma Separated Value) format. There are two ways to organize the numeric values in the data file. If you use the COLUMN_MAJOR format, the first column represents the x-values and subsequent columns represent the y-values, where each column is a group. If you use the ROW_MAJOR format, the first row represents the x-values and subsequent rows represent the y-values, where each row is a group. Use the **CSV.SetOrientation** method to initialize the csv argument for the proper data orientation.

```
[C#]
public ElapsedTimeGroupDataset(
   CSV csv,
   StorageFile sfile,
   int rowskip,
   int columnskip
);
```

csv	An instance of a **CSV** object.
sfile	A StorageFile object representing the input file
rowskip	Skip this many rows before starting the read operation.
columnskip	For each row of data, skip this many columns before reading the first value from the row.

The only supported format for elapsed time values in a CSV file is d.hh.mm.ss.fff, (3.14:23:12.333 as an example of an elapsed time of three days, 14 hours, 23 minutes, 12 seconds and 333 milliseconds).

You can also modify a point at a time using **SetElapsedTimeXDataValue** (or **SetElapsedTimeYDataValue**) if you are using **TimeSpan** objects**,** and **SetYDataValue** or **SetXDataValue**) if you use millisecond values. If you need to add new points to the dataset, increasing its size, use one of the **AddDataPoint**, or **InsertDataPoint** methods. Delete data points using the **DeleteDataPoint** method. In order to see the modified dataset, force the graph to redraw using **ChartView.UpdateDraw** method.

Example of creating a group datasets from numeric arrays

[C#]

```
int nNumPnts = 5, nNumGroups = 4;
TimeSpan[] xValues = new TimeSpan[nNumPnts];
```

```
double[,] groupBarData = new double[nNumGroups, nNumPnts];

xValues[0] = TimeSpan.FromMinutes(1);
groupBarData[0, 0] = 6.3; groupBarData[1, 0] = 3.1;
groupBarData[2, 0] = 2.2; groupBarData[3, 0] = 1.8;

xValues[1] = TimeSpan.FromMinutes(2);
groupBarData[0, 1] = 5.8; groupBarData[1, 1] = 4.3;
groupBarData[2, 1] = 2.8; groupBarData[3, 1] = 1.5;

xValues[2] = TimeSpan.FromMinutes(3);
groupBarData[0, 2] = 5.5; groupBarData[1, 2] = 4.5;
groupBarData[2, 2] = 2.5; groupBarData[3, 2] = 2.1;

xValues[3] = TimeSpan.FromMinutes(4);
groupBarData[0, 3] = 4.1; groupBarData[1, 3] = 5.4;
groupBarData[2, 3] = 4.1; groupBarData[3, 3] = 3.2;

xValues[4] = TimeSpan.FromMinutes(5);
groupBarData[0, 4] = 3.8; groupBarData[1, 4] = 5.6;
groupBarData[2, 4] = 4.3; groupBarData[3, 4] = 3.3;

ElapsedTimeGroupDataset Dataset1 =
    new ElapsedTimeGroupDataset("ElapsedTimeGroupData", xValues, groupBarData);
```

Read data from a file using the ElapsedTimeGroupDataset ReadElapsedTimeGroupDataset method, passing in a StorageFolder object, and a file name. The method containing the ReadElapsedTimeGroupDataset or WriteElapsedTimeGroupDataset calls must be market *async*.

Example of reading and writing a elapsed time group dataset from a CSV file

[C#]

```
    StorageFolder dataFolder = Windows.Storage.ApplicationData.Current.LocalFolder;


    int result = 0;
    result = await Dataset1.WriteElapsedTimeGroupDataset(dataFolder,"Dataset11.csv");
    result = await Dataset2.WriteElapsedTimeGroupDataset(dataFolder,"Dataset12.csv");
    result = await Dataset3.WriteElapsedTimeGroupDataset(dataFolder,"Dataset13.csv");
    .
    .
    .
    // Read it back in just as a test
    result = await Dataset1.ReadElapsedTimeGroupDataset(dataFolder,"Dataset11.csv");
    result = await Dataset2.ReadElapsedTimeGroupDataset(dataFolder,"Dataset12.csv");
    result = await Dataset3.ReadElapsedTimeGroupDataset(dataFolder,"Dataset13.csv");
```

Event Group Dataset

Class EventGroupDataset

ChartObj

|

+--ChartDataset

|

+--GroupDataset

|

+--EventGroupDataset

The **EventGroupDataset** class is the group dataset version of EventSimpleDataset. See the background information under the EventSimpleDataset. It is used to supply data to the group plotting classes: OHLC, Candlestick, GroupBar, Stacked Bar, etc..

A ChartEvent can have one or more y-values. In the case of an EventSimpleDataset, usually it will have a single value, as seen in the EventSimpleDataset programming example, one y-value for each x-value. In the EventGroupDataset, each ChartEvent object can have multiple y-values for each x-value, as seen in the programming example below..

EventGroupDataset constructors

```
[C#]
public EventGroupDataset(
   string sname,
   ChartEvent [] ev
);
```

sname Specifies the name of the dataset.

ev An array of ChartEvent objects.

Create an array of ChartEvent objects, specifying the time-stamp and y-values for each ChartEvent object and use that to initialize a EventSimpleDataset.

The next constructor creates an EventGroupDataset using the x- and y-values stored in a file that uses the CSV (Comma Separated Value) format. Only the COLUMN_MAJOR format is supported, where each row presents a ChartEvent object, and the columns are organized as: description, short description, x-axis string label, tool tip string, position, time stamp, numeric time stamp, y-value index (index for the y-value to use when the ChartEvent contains multiple y-values), and the y-values.

```
[C#]
public EventGroupDataset(
   CSV csv,
   StorageFile sfile,
   int rowskip,
   int columnskip
);
```

csv An instance of a **CSV** object.

sfile A StorageFile object representing the input file

rowskip Skip this many rows before starting the read operation.

columnskip For each row of data, skip this many columns before starting this read operation.

A **DateTimeFormatInfo** object, and a date time format string, in the **CSV** class, control the interpretation of the **ChartCalendar** values. The format in the file must match the format specified for the **CSV** class. The underlying conversion mechanism calls the **DateTime.ToString(String formatstring, DateTimeFormatInfo info)** method for the conversion. The default format for the date time *formatstring* object is "M/dd/yy". Call the **SetDateTimeFomatString** method to change the default date time format. See the documentation for the .Net **DateTime.ToString** method to figure out the various formatting options for the date time format string. If you are into internationalization (and difficult to understand .Net documentation), you can also create your own **DateTimeFormatInfo** object**,** installing it in the **CSV** object using **CSV.SetTimeDateFormat** method**.** The date time format string and the **DateTimeFormatInfo** object apply to both CSV files used for input, and CSV files used for output. If an attempt is made to read date/time values that do not match the desired format, the data values are set to invalid date/time values.

You can also modify a point at a time using **SetEvent.** If you need to add new points to the dataset, increasing its size, use one of the **AddEvent**, or **InsertEvent** methods. Delete data points using the **DeleteEvent** method. In order to see the modified dataset, force the graph to redraw using **ChartView.UpdateDraw** method.

.

Example of creating a simple event datasets, extracted from the ChartEventExamples.CandlestickEventChart example program.

[C#]

```
double minval = 0.0, maxval = 0.0;
int incrementbase = ChartObj.MINUTE;
int increment = 10;
ChartCalendar currentdate = new ChartCalendar();
ChartEvent[] eventArray = new ChartEvent[nNumPnts];

ChartEvent currentEvent = new ChartEvent();
for (i = 0; i < nNumPnts; i++)
{
    double position = i + 1;
    if (i == 0)
    {
       currentdate = ChartCalendar.CalendarDaysAdd(currentdate, 1, weekmode);
       currentdate.SetTOD(9, 33, 0);
       xValues[0] = (ChartCalendar)currentdate.Clone();
       currentdate = ChartCalendar.CalendarDaysAdd(currentdate, 1, weekmode);
       stockPriceData[3] = 25;  // close
       stockPriceData[0] = 25;   // open
       stockPriceData[1] = 26;   // high
       stockPriceData[2] = 24;   // low
       currentEvent = new ChartEvent(xValues[0], 1, stockPriceData);
       currentEvent.AxisLabel = "XXX" + "1";
       eventArray[0] = currentEvent;
     }
     else
     {
        xValues[i] = (ChartCalendar)currentdate.Clone();
        stockPriceData[3] += 2 * (0.5 - ChartSupport.GetRandomDouble()); // close
        stockPriceData[0] += 2 * (0.5 - ChartSupport.GetRandomDouble()); // open
```

```
        minval = Math.Min(stockPriceData[3], stockPriceData[0]);
        maxval = Math.Max(stockPriceData[3], stockPriceData[0]);
       stockPriceData[1] = maxval + 1.5 * ChartSupport.GetRandomDouble();  // high
       stockPriceData[2] = minval - 1.5 * ChartSupport.GetRandomDouble();  // low
        currentdate.Add(incrementbase, increment);
        if (currentdate.Get(ChartObj.HOUR_OF_DAY) >= 16)
        {
            currentdate.Add(ChartObj.DAY_OF_YEAR, 1);
            currentdate.SetTOD(9, 30, 0);
        }


       currentEvent = new ChartEvent(xValues[i], position, stockPriceData);
       currentEvent.AxisLabel = "XXX" + position.ToString();
       currentEvent.ToolTip = "ToolTip" + position.ToString();
       eventArray[i] = currentEvent;
    }
}

EventGroupDataset Dataset1 = new EventGroupDataset("Stock Data", eventArray, 4);
EventSimpleDataset Dataset2 = Dataset1.ConvertToEventSimpleDataset(1);

EventCoordinates pTransform1 = new EventCoordinates(Dataset1);
```

Read data from a file using the EventGroupDataset ReadEventGroupDataset method, passing in a StorageFolder object, and a file name. The method containing the ReadEventGroupDataset or WriteEventGroupDataset calls must be market *async*.

Example of reading and writing a event group dataset from a CSV file

[C#]

```
    StorageFolder dataFolder = Windows.Storage.ApplicationData.Current.LocalFolder;


    int result = 0;
    result = await Dataset1.WriteEventGroupDataset(dataFolder,"Dataset11.csv");
    result = await Dataset2.WriteEventGroupDataset(dataFolder,"Dataset12.csv");
    result = await Dataset3.WriteEventGroupDataset(dataFolder,"Dataset13.csv");
    .
    .
    .
    // Read it back in just as a test
    result = await Dataset1.ReadEventGroupDataset(dataFolder,"Dataset11.csv");
    result = await Dataset2.ReadEventGroupDataset (dataFolder,"Dataset12.csv");
    result = await Dataset3.ReadEventGroupDataset (dataFolder,"Dataset13.csv");
```

4. Scaling and Coordinate Systems

ChartScale
- **LinearScale**
- **LogScale**
- **TimeScale**
- **ElapsedTimeScale**
- **EventScale**

UserCoordinates
- **WorldCoordinates**
  - **WorkingCoordinates**
    - **PhysicalCoordinates**
      - **CartesianCoordinates**
        - **PolarCoordinates**
        - **AntennaCoordinates**
        - **EventCoordinates**
      - **TimeCoordinates**
      - **ElapsedTimeCoordinates**

The starting point for all drawing in a window is the .Net 2D device coordinate system. The coordinate system uses a default device resolution of the underlying .Net window, regardless of the output device. A .Net window maintains a viewport for the client area of the window, controlling the position and size of the drawing area in the window. Graphics output is clipped to the viewport, preventing graphics output in one window from over-writing graphics in another window. The user coordinate system for the window starts at (0,0) in the upper left corner and extends in the positive direction down and to the right.

Plot area and graph area

The *plot area* of a graph is the area where the plot data objects (line plots, bar plots, etc.) are drawn. The *graph area* is the entire area of the chart window. The graph area includes the plot area as a subset. Usually, the plot area is smaller than the graph area and resides roughly centered in the graph area. The border around the plot area is sized large enough to display the axis tick mark labels, axis titles, legends, chart titles, footers, and any other object in the graph. Create a physical coordinate system for a chart, and you are setting the minimum and maximum values for the x and y dimensions of the plot area.

Most chart objects require access to the chart coordinate system for proper positioning in the chart window. Some chart objects, axis objects in particular, often reside on the edge or outside of the plot area. Important parts of the axis, the tick marks and tick mark labels, are usually outside of the plot area. The tick marks and tick mark labels must align perfectly with the coordinate system inside the plot area.

There are many different techniques to align the coordinate system inside the plot area with the coordinate system used in drawing chart objects outside of the plot area. One technique is to maintain the physical coordinate system inside the plot area, and use a normalized coordinate system for the graph area. Whenever a chart object in the graph area, a y-axis tick mark for example, needs to be aligned with the coordinate system inside the plot area, the software converts the tick mark placement value from physical coordinates to normalized coordinates using standardized coordinate conversion routines. The drawback of this technique is what I will call the "odd pixel problem". The odd pixel problem shows up when you try map physical, normalized and user coordinate systems based on floating point numbers onto a pixel coordinate system using an integer coordinate system. Unless the corners of the plot area fall on exact pixel boundaries, converting from plot area coordinates to graph area coordinates, once translated to pixels, can be up to one pixel off.

The alternative technique, used in this software library, is to use a single coordinate system. The physical coordinate system defined for the plot area is extended in all four directions – left, right, top and bottom. It is extended so that the physical coordinates of the four corners of the plot area remain unchanged. The four corners of the graph area are assigned calculated, physical coordinate values so that when it is overlaid on to the plot area there is an exact 1:1 correspondence for all points inside the plot area. Once this calculation is made, there is no need to use the physical coordinate system assigned to the plot area. Instead, the physical coordinate system of the graph area is used instead. Chart axes objects and plotted data always align because they are plotted using the exact same physical coordinate system. The only difference is that plotted data is clipped to the plot area while axes objects are not clipped.

For example, assume a graph area with the dimensions of 400x400 units, and a plot area with the dimensions 200x200 centered inside the graph area. This implies that there is a 100 unit boundary around all four sides of the plot area. The desired chart uses a physical coordinate system of (0, 0,100,100). These coordinates apply to the plot area. Instead of using the plot area coordinate system, the coordinate system (-50,-50,150,150) is calculated and used to scale the graph area. This does not guarantee that any point plotted in plot area coordinate system will always map to the exact same pixel as the same point plotted in the graph coordinate system, the odd pixel problem still exists. We avoid the odd pixel problem by never plotting points using the plot area coordinate system, using only the graph area coordinate system instead. The calculated physical coordinate system applied to the graph area is referred to as the *working* coordinate system.

Coordinate Systems

A **QCChart2D for Windows Apps** library uses other coordinate systems mapped onto the default UWP device independent coordinate system. These other coordinate systems include world coordinates, working coordinates, and physical coordinates.

User Coordinates

The **UserCoordinates** class manages a simple viewport drawing system using the .Net UWP drawing classes.

World Coordinates

The **WorldCoordinates** class maps a linear, double based, coordinate system onto the integer based user coordinate system of the **UserCoordinates** class. Where the underlying user coordinate system may have an integer range (for example 0-400, 0-300 units), the world coordinate system is able to map this to a completely arbitrary, double based, linear range (for example (0.0 to 10.0, 0.0 to 10.0) . The world coordinate system applies to the entire graph window, and not just the plot area.

Working Coordinates

The **WorkingCoordinates** class manages a working coordinate system that maps the physical coordinate system of the plot area into a linear, world coordinate system applied to the whole viewport. For example, if the desired chart plot area uses a physical coordinate system of (0.0, 0.0, 100.0, 100.0) and the plot area is centered in the graph area with the plot area ½ the width and height of the graph area, then the coordinate system (-50, -50, 150, 150) is calculated and used to scale the graph area. The **WorkingCoordinates** class uses the underlying **WorldCoordinates** class to scale the viewport to the final world coordinates scale.

Physical Coordinates

The **PhysicalCoordinates** abstract class is responsible for mapping the plot area coordinate system (whether it is linear, logarithmic, date/time, polar, antenna, continuous or discontinuous) into a continuous linear coordinate system. It uses the **WorkingCoordinates** class to map this plot area coordinate system to the entire viewport. The **PhysicalCoordinates** system uses independent scale objects, derived from **ChartScale**, to manage coordinate conversions for the x- and y-dimensions. This way the x-coordinate can use one coordinate conversion object (**LinearScale**, **LogScale**, **TimeScale**) and the y-coordinate another.

There are six concrete implementations of the **PhysicalCoordinates** class: **CartesianCoordinates**, **TimeCoordinates**, **ElapsedTimeCoordinates**, **EventCoordinates**, **PolarCoordinates** and **AntennaCoordinates**. Use the **CartesianCoordinates** class for any combination of linear and logarithmic scaling for the x- and y-coordinate. Use the **TimeCoordinates** class when you want a time/date scale for the x-coordinate and a linear or logarithmic scale for the y-coordinate. Use the **EventCoordinates** if you have discontinuous, or irregular, time, suchs as that found in worldwide financial markets. Use the **ElapsedTimeCoordinates** class when you want a elapsed time scale (no date information) for the x-coordinate and a linear or logarithmic scale for the y-coordinate. Use the **PolarCoordinates** for polar coordinates where the magnitude coordinate is linear and the polar angle coordinate extends from 0 to 360 degrees (or 0 to 2*pi radians) counter-clockwise, starting at 3:00. Use the **AntennaCoordinates** for antenna coordinates where the radius value is linear and the angle coordinate extends from 0 to 360 degrees clockwise, starting at 12:00.

Normalized coordinates

Normalized coordinates are a special case of linear physical coordinates, where the linear physical scale (0.0 - 1.0, 0.0 – 1.0) is applied to either the graph area, or the plot area of the chart. ***Graph normalized coordinates*** maps the upper left corner of the graph window to the xy coordinates (0.0,0.0) and the lower right corner of the graph area to the xy coordinate (1.0,1.0). ***Plot normalized coordinates*** maps the upper left corner of the plot area to the xy coordinates (0.0,0.0) and the lower right corner of the plot area to the xy coordinates (1.0,1.0).

A copy of a concrete instance of the **PhysicalCoordinates** class is stored in every **GraphObj** derived object. This includes all axes, plot objects, text objects and data markers. When a chart object draws itself, it uses the viewport and the physical coordinate system stored in its instance of a **PhysicalCoordinates** class. A chart can have one or more instances of the **PhysicalCoordinates** since a single chart can plot data against one or more physical coordinate systems.

Important numeric considerations

Value limiting

A chart should be scaleable to any numeric range that a user wants to plot. This incorporates the entire range of floating point numbers support under .Net. A range of +-10^-30 is just as valid as a range of +-10^30, even though there is 60 orders of magnitude of difference. A user can attempt to plot data with an extremely large dynamic range (the +-10^30 range) in a chart scaled for an extremely small range (the +-10^-30 range). The resulting user coordinate values resulting from such an extreme case can easily exceed the numeric range supported by the plotting functions.

Bad value checking

Invalid data often finds its way into chart. Invalid data can take many different forms. The most obvious is the introduction of numeric values that do not fit the .Net floating point format. These types of numbers are often found in databases and representing non-initialized or improperly initialized data. .Net cannot include an invalid floating point number in a calculation, so it is best to try and avoid them. Another type of invalid data is data that is a valid floating point number, but is never less considered invalid by the user. Often when data is outside of a predetermined range, it is invalid. Mark a data value in a dataset invalid using the **ChartDataset.SetValidData** method. If a data value equals Double.MAX_VALUE, it is also considered invalid.

Taking the logarithm of 0 or a negative number

If a charting package is capable of logarithmic and semi-logarithmic plotting, it must be protected against the error condition of taking the log of any number <= 0.0. This often happens when a chart is initially setup with a linear scale, with a minimum physical coordinate value of 0.0 and a maximum coordinate value equal to some large number. The user changes to a logarithmic scale, but forgets to change the minimum coordinate value of the scale from 0.0 to some positive non-zero number. The coordinate conversion routines will halt the first time the log(0.0) is in a calculation. The same is also true if the minimum coordinate value is any negative number. This software always checks for this condition and changes the minimum coordinate value using the following criteria. If the minimum coordinate value for a logarithmic scale is less than or equal to 0.0 it is assumed that the user made an error and coordinate value is

set to 1.0. If the minimum coordinate value is greater than 0.0 but less than the value of MIN_LOG_VALUE, the minimum coordinate value is set to MIN_LOG_VALUE.

Positioning the Plot Area in Graph Area

The **WorkingCoordinates** class has a group of methods - **SetGraphBorderFrame**, **SetGraphBorderDiagonal**, and **SetGraphBorderInsets** - that position the plot area of the chart in the graph viewport. Since the coordinate system scaling classes are subclasses of **WorkingCoordinates**, these methods are part of those classes. These methods are redundant and only one need be called. The default position of the plot area in the graph view port is at x = 0.2, y= 0.2, width = 0.6, height = 0.6, specified using graph normalized coordinates.

This method initializes the position and size of the plot area inside the graph area, specified using a rectangle to specify graph normalized coordinates.

SetGraphBorder methods

```
[C#]
public void SetGraphBorderFrame(
   Rectangle2D border
);
```

border Specifies the rectangle defining the plot area border.

This method initializes the position and size of the plot area inside the graph area, specified using graph normalized position and size values.

```
[C#]
public void SetGraphBorderFrame(
   double rLeft,
   double rTop,
   double width,
   double height
);
```

where

rLeft	The left x-position of the plot area inside the graph area specified using graph normalized coordinates.
rTop	The top y-position of the plot area inside the graph area specified using graph normalized coordinates.
width	The width of the plot area inside the graph area specified using graph normalized coordinates.
height	The height of the plot area inside the graph area specified using graph normalized coordinates.

This method initializes the size and position of the plot area inside the graph area, specified using graph normalized values for the opposite corners of the region.

```
[C#]
public void SetGraphBorderDiagonal(
   double rLeft,
   double rTop,
   double rRight,
   double rBottom
);
```

rLeft	The left x-position of the plot area inside the graph area specified using graph normalized coordinates.
rTop	The top y-position of the plot area inside the graph area specified using graph normalized coordinates.
rRight	The right x-position of the plot area inside the graph area specified using graph normalized coordinates.
rBottom	The bottom y-position of the plot area inside the graph area specified using graph normalized coordinates.

This method initializes the insets of the plot area inside the graph area, specified using graph normalized values.

```
[C#]
public void SetGraphBorderInsets(
   double rLeft,
   double rTop,
   double rRight,
   double rBottom
);
```

rLeft The left inset of the plot area inside the graph area specified using graph normalized coordinates.

rTop The top inset of the plot area inside the graph area specified using graph normalized coordinates.

rRight The right inset of the plot area inside the graph area specified using graph normalized coordinates.

rBottom The bottom inset of the plot area inside the graph area specified using graph normalized coordinates.

The following examples all position the plot area of the chart in the upper right quadrant of the graph viewport.

[C#]

```
CartesianCoordinates simpleScale;
simpleScale = new CartesianCoordinates(xMin, yMin, xMax, yMax);

// Use ONE of the example below
// Example #1
simpleScale.SetGraphBorderFrame(new Rectangle2D(0.5, 0.0, 0.5, 0.5));

// Example #2
simpleScale.SetGraphBorderFrame(0.5, 0.0, 0.5, 0.5);

// Example #3
simpleScale.SetGraphBorderDiagonal (0.5, 0.0, 1.0, 0.5);

// Example #4
simpleScale.SetGraphBorderInsets (0.5, 0.0, 0.0, 0.5);
```

Linear and Logarithmic Coordinate Scaling

Class CartesianCoordinates

PhysicalCoordinates

|

+-- CartesianCoordinates

The **CartesianCoordinates** class scales the chart plot area for a physical coordinate system that uses linear and/or logarithmic scaling.

There are three main ways to scale the plot area:

- Scale the minimum and maximum x- and y- values explicitly
- Use an auto-scale method that calculates appropriate minimum and maximum x- and y-values based on the x- and y-values in one or more datasets
- Use a combination of the first two methods. It is useful to be able to run an auto-scale function, and then change the minimum or maximum value of one or more coordinate endpoints.

Linear Coordinate Scaling

The default coordinate system for the **CartesianCoordinates** class is linear for both x and y. If you already know the range for x and y for the plot area, you can scale the plot area explicitly.

The example below uses a **CartesianCoordinates** constructor to initialize the coordinates to the proper values.

CartesianCoordinates constructor with explicit scaling

[C#]

```
double xMin = -5;
double xMax = 15;
double yMin = 0;
double yMax =  105;

CartesianCoordinates simpleScale;
simpleScale = new CartesianCoordinates(xMin, yMin, xMax, yMax);
```

Another technique uses the default constructor and scales the coordinates using the **CartesianCoordinates.SetCoordinateBounds** method.

Example of explicit scaling of a CartesianCoordinates object using the CartesianCoordinates.SetCoordinateBounds method

[C#]

```
double xMin = -5;
double xMax = 15;
double yMin = 0;
double yMax =  105;
CartesianCoordinates simpleScale = new CartesianCoordinates();

simpleScale.SetCoordinateBounds(xMin, yMin, xMax, yMax);
```

It is possible to scale the bounds of the coordinate system based on the data values in a dataset. There are constructors and methods that take a single dataset and others that take an array of datasets.

Example of auto-scaling a CartesianCoordinates object using a single dataset

[C#]

```
double [] xData = {1,2,3,4,5,6,7,8,9,10};
double [] yData = {10, 22, 33, 44, 55, 46, 33, 25, 14, 9};

SimpleDataset dataset = new SimpleDataset("Sales", xData, yData);
CartesianCoordinates simpleScale  = new CartesianCoordinates();
simpleScale.AutoScale(dataset);
```

You can control the "tightness" of the auto-scale values about the dataset values using other versions of the **CartesianCoordinates.AutoScale** method that take rounding mode parameters.

Example of auto-scaling a CartesianCoordinates object using a single dataset and explicit rounding mode parameters

```
simpleScale.AutoScale(dataset,  ChartObj.AUTOAXES_FAR, ChartObj.AUTOAXES_FAR)
```

You can auto-scale the bounds of the coordinate system using a dataset, and then explicitly modify the range the auto-scale selected. There are methods that set the minimum and maximum values of the x- and y-scales. This way you can use the auto-scale methods for the values of one scale (the y-scale in the example below), but explicitly set the values for the other scale (the x-scale in the example below).

Example of modifying the minimum and maximum values selected by an auto-scale method.

[C#]

```
double [] xData = {2,3,4,5,6,7,8,9};
double [] yData = { 22, 33, 44, 55, 46, 33, 25, 14};

SimpleDataset dataset = new SimpleDataset("Sales", xData, yData);
CartesianCoordinates simpleScale  = new CartesianCoordinates();
simpleScale.AutoScale(dataset);
simpleScale.SetScaleStopX(10);
simpleScale.SetScaleStartX(1.0);
```

The auto-scale methods that use an array of datasets to determine the proper range are very similar.

Example of auto-scaling a CartesianCoordinates object using the multiple datasets

[C#]

```
double [] xData1 = {1,2,3,4,5,6,7,8,9,10};
double [] yData1 = {10, 22, 33, 44, 55, 46, 33, 25, 14, 9};
double [] xData2 = {10,9,8,7,6,5,4,3,2,1};
double [] yData2 = {20, 12, 43, 54, 15, 26, 63, 25, 24, 19};
double [] xData3 = {5,6,7,6,5,4,5,6,7,8};
double [] yData3 = {30, 52, 13, 64, 25, 76, 13, 35, 24, 19};

SimpleDataset dataset1 = new SimpleDataset("Sales1",xData1,yData1);
SimpleDataset dataset2 = new SimpleDataset("Sales2",xData2,yData2);
SimpleDataset dataset3 = new SimpleDataset("Sales3",xData3,yData3);
SimpleDataset [] datasetsArray = new SimpleDatasets[3];
datasetsArray[0] = dataset1;
datasetsArray[1] = dataset2;
datasetsArray[2] = dataset3;
CartesianCoordinates simpleScale  = new CartesianCoordinates();
simpleScale.AutoScale(datasetsArray);
```

There is a version of the multiple dataset auto-scale routine that also specifies rounding mode parameters.

```
simpleScale.AutoScale(datasetsArray,ChartObj.AUTOAXES_FAR, ChartObj.AUTOAXES_FAR)
```

Logarithmic Coordinate Scaling

The previous examples assume that both the x- and y- scales are linear. If the x and/or y scale are to be logarithmic, then use the **CartesianCoordinates** constructor that has scale mode parameters.

Example of explicit scaling of three different logarithmic CartesianCoordinates objects

[C#]

```
double xMin = 1;
double xMax = 1000;
double yMin = 0.2;
double yMax =  2000;
CartesianCoordinates logYScale =
          new CartesianCoordinates(ChartObj.LINEAR_SCALE, ChartObj.LOG_SCALE);
logYScale.SetCoordinateBounds(xMin, yMin, xMax, yMax);

CartesianCoordinates logXScale =
          new CartesianCoordinates(ChartObj.LOG_SCALE, ChartObj.LINEAR_SCALE);
logXScale.SetCoordinateBounds(xMin, yMin, xMax, yMax);

CartesianCoordinates logXLogYScale =
          new CartesianCoordinates(ChartObj.LOG_SCALE, ChartObj.LOG_SCALE);
logXLogYScale.SetCoordinateBounds(xMin, yMin, xMax, yMax);
```

Note: When you explicitly scale the minimum and maximum values for a scale set to logarithmic coordinates, make sure you use valid values, i.e. non-negative values greater than 0.0.

The auto-scale routines work for both linear and logarithmic scales. If you use the auto-scale methods, it is important that you call the auto-scale methods **after** you establish if a scale is

linear or logarithmic. This is because the auto-scale routines will allow zero and negative values for the minimum and maximum of a linear scale, but not a logarithmic scale. If you call the auto-scale routines first, while the scales are set to the default linear scale mode, and change one or both of the scales to logarithmic mode, you can introduce invalid negative and zero values into the logarithmic coordinate system.

Example of auto-scaling a CartesianCoordinates object that has a logarithmic y-scale, using a single dataset

[C#]

```
double [] xData = {2,3,4,5,6,7,8,9};
double [] yData = { 2, 33, 440, 5554, 46123, 332322, 5435641, 64567551};

SimpleDataset dataset = new SimpleDataset("Sales", xData, yData);
CartesianCoordinates simpleScale =
              new CartesianCoordinates(ChartObj.LINEAR_SCALE, ChartObj.LOG_SCALE);
simpleScale.AutoScale(dataset);
simpleScale.SetScaleStopX(10);
simpleScale.SetScaleStartX(1.0);
```

Coordinate Systems using times and dates

Many charting applications use data where the x- or y-coordinate values are in the form of time and date records. The creation of a powerful yet flexible time and date physical coordinate system for scaling charts is a major challenge. Important design goals include:

- The programmer scales the graph using objects of the **ChartCalendar** class.
- The scale should support a continuous range of date/time scaling from milliseconds to hundreds of years.
- The scale should have a 5/7 day week option.
- A day does not have to be 24 hours long; instead, it can have a specific range, for example: 8:30 AM to 4:00 PM.
- Time and date axes, used to delineate a date/time scale, must be able to display and label tick marks for days, weeks, months and years, taking into account the non-uniformity of years, months, weeks and days in the Gregorian calendar.

The ChartCalendar Class

The **ChartCalendar** class represents time and date information in the format used by the majority of the world. It is a universal time class, capable of representing time with a resolution of milliseconds and a dynamic range of hundreds of years. It contains time and date fields that

specify an exact moment in time, i.e. November 7, 2000 20:43:22.554. Since the **ChartCalendar** class represents both time of day and calendar dates, is not necessary to use separate time of day and date classes to manage data collected every few seconds, yet spans a day or more. An example of this type of date/time range is a graph of experimental data, sampled every 15 seconds, starting on June 12, 2001 11:43:00 PM and continuing until June 15, 2001 1:03:45 AM.

The major application for date/time chart scaling on the Internet is the display of financial market data. Multiply the number of on-line investors using the charting tools of on-line brokerage firms and financial web sites by the number of individual stocks, bonds, options and futures contracts that they are able to chart and it becomes apparent that the number of potential charts is almost infinite. Many stocks have more than 50 years of hourly historical information associated with them. A chart comparing the relative performance of General Electric, IBM, Dupont, Ford and Kodak, compared to the Dow Jones Industrial average, since the dawn of the computer age in 1950, will quickly highlight the above average stock market gains that can be made by investing in the right stocks. A user may start by looking at a fifty-year window on a stock, then through successive zoom operations narrow the window to two years, one month, one day, one hour and perhaps even one minute.

5 vs. 7 Day Work Weeks

Historically, western cultures use a five-day workweek. Much of the data suitable for charting includes data for only the workdays of Monday through Friday, excluding weekends. Data associated with financial markets is the most common. Stocks trade on Monday through Friday, excluding weekends and holidays. In the display of financial market information, it is not useful, and in many cases misleading, to scale a graph, using a seven-day work week, and then only plot data using five of the seven days. The gaps left in the chart waste valuable chart space. For line plots, if a line is drawn from the last data point on Friday to the first data point on Monday, the result gives a visual impression that the trend from Friday to Monday lasted two days (Friday midnight to Sunday midnight), when in fact it may have only lasted minutes. The **QCChart2D for Windows Apps** date/time scaling, axes and auto-axes classes support dropping weekends, as an option, from the scale. One minute after Friday midnight is 12:01 AM Monday morning.

Non-24 Hour Days

Similarly, a full day of time stamped data may not fill an entire 24 hour day. Financial markets have specific trading hours. For example, the NYSE is open from 9:30 AM EST to 4:00 PM EST. Stocks traded on the NYSE will have a time stamp in this time range. Plotting NYSE stock data for several days, using data points sampled every fifteen minutes, will result in large gaps in a traditional chart, where 2/3 of the day stock trading is inactive. It is possible to treat the unused portion of the day in a manner analogous to weekends. It is possible to eliminate unused hours and minutes of a day from the chart coordinate system. The **QCChart2D for Windows Apps** coordinate system allows the programmer to specify a starting and ending time for a days worth of data. In the NYSE stock market example the starting time is 9:30 AM EST and the ending

time is 4:00 PM EST. Any data outside of this range is invalid and not plotted. In terms of the resulting chart, one minute after 4:30 PM is 9:31 AM. Combine the starting and ending time parameters, with the option of deleting weekends from consideration, and one minute after 4:30 PM Friday is 9:31 AM Monday.

Non-Uniformity of Date/Time Tick Marks

There even more complications associated with date/time scales. The axis that delineates a date/time scale must take into account the non-uniform nature of date/time tick marks and tick mark labels. A month has a variable number of days. When months are used as the major tick mark interval, and days are the minor tick mark interval, the software must be capable of plotting 28, 29, 30 or 31minor tick marks (days) for every major tick mark (months), depending on the month. Another example is the use of months as the major tick mark interval, and weeks as the minor tick mark interval. Some months will have four minor tick marks (start of each new week) while others will have five. The software also needs to take into the variable number of days/year due to leap years. Date/time axis tick marks and labels become even more complicated when the 5-day/7-day option and the working hours/day options are used.

The **QCChart2D for Windows Apps** library uses milliseconds as the underlying time base for all date/time coordinate system. When a scale is created using two **ChartCalendar** dates as end points, the software calculates the number of milliseconds seconds between the starting date and the ending date. If the coordinate system is based on a 5-day week, the milliseconds associated with the missing weekends are not counted. If the coordinate system does not use 24 hours a day, the milliseconds associated with the missing part of the day are not counted. A linear coordinate system is scaled using the range of calculated milliseconds. Data points plotted in this coordinate system have their date/time value converted to milliseconds seconds by subtracting the starting date of the scale from the data point date, making sure to exclude the seconds associated with weekends and fractional days, if necessary. The data point is plotted in the milliseconds based linear coordinate system. Since the **QCChart2D for Windows Apps** library uses milliseconds as the underlying time base, the minimum allowable displayable range is one millisecond. For ranges smaller than one second, the programmer needs to convert the **ChartCalendar** values to seconds or milliseconds and use the **CartesianCoordinates** class to scale the chart. You loose the date/time formatting of the axis labels, but this should not matter if you are dealing in the sub millisecond realm.

Class TimeCoordinates

PhysicalCoordinates

|

+-- **TimeCoordinates**

The **TimeCoordinates** class scales the chart plot area for physical coordinate systems that use date/time scaling. The basic techniques are essentially the same as those used with linear and logarithmic scaling; only the **TimeCoordinates** class uses **ChartCalendar** dates in place of numeric values for the x- or y-axis scale values. The minimum and maximum values of the x- and y-scales can be set explicitly, set using one of the auto-scale methods, or set using a combination of the two.

The default coordinate system for the **TimeCoordinates** class is time for the x-scale and linear for the y-scale scale. The y-scale can be logarithmic and you can set that mode using the explicit scale mode version of the **TimeCoordinates** constructor. If you already know the range for x and y for the plot area, you can scale the plot area explicitly. In the example below a **TimeCoordinates** constructor initializes the coordinates to the proper values.

TimeCoordinates constructor with explicit scaling of a time based x-scale, and a numeric based y-scale.

[C#]

```
ChartCalendar xMin = new ChartCalendar(1996, ChartObj.FEBRUARY,5);
ChartCalendar xMax = new ChartCalendar(2002, ChartObj.JANUARY,5);
double yMin = 0;
double yMax =  105;

TimeCoordinates simpleTimeScale;
simpleTimeScale = new TimeCoordinates(xMin, yMin, xMax, yMax);
```

TimeCoordinates constructor with explicit scaling of a numeric based x-scale, and a time based y-scale.

[C#]

```
double xMin = 0;
double xMax =  105;
ChartCalendar yMin = new ChartCalendar(1996, ChartObj.FEBRUARY,5);
ChartCalendar yMax = new ChartCalendar(2002, ChartObj.JANUARY,5);

TimeCoordinates simpleTimeScale;
simpleTimeScale = new TimeCoordinates(xMin, yMin, xMax, yMax);
```

Another technique uses the default constructor and scales the coordinates using the **TimeCoordinates.SetTimeCoordinateBounds** method.

Example of explicit scaling of a TimeCoordinates object (time x-scale and numeric y-scale) using the TimeCoordinates.SetTimeCoordinateBounds method

[C#]

```
ChartCalendar xMin = new ChartCalendar(1996, ChartObj.FEBRUARY,5);
ChartCalendar xMax = new ChartCalendar(2002, ChartObj.JANUARY,5);
double yMin = 0;
double yMax =  105;
TimeCoordinates simpleTimeScale = new TimeCoordinates();
```

```
simpleTimeScale.SetTimeCoordinateBounds (xMin, yMin, xMax, yMax);
```

Example of explicit scaling of a TimeCoordinates object (numeric x-scale and time y-scale) using the TimeCoordinates.SetTimeCoordinateBounds method

[C#]

```
double xMin = 0;
double xMax =  105;
ChartCalendar yMin = new ChartCalendar(1996, ChartObj.FEBRUARY,5);
ChartCalendar yMax = new ChartCalendar(2002, ChartObj.JANUARY,5);
TimeCoordinates simpleTimeScale = new TimeCoordinates();

simpleTimeScale.SetTimeCoordinateBounds (xMin, yMin, xMax, yMax);
```

It is possible to scale the bounds of the coordinate system based on the data values in a time-based dataset, **TimeSimpleDataset** and **TimeGroupDataset**. There are constructors and methods that take a single dataset and others that take an array of datasets.

Example of auto-scaling a TimeCoordinates object using a single dataset

[C#]

```
ChartCalendar [] xData = { new ChartCalendar(1996, ChartObj.FEBRUARY,  5),
                           new ChartCalendar(1996, ChartObj.MARCH,  5),
                           new ChartCalendar(1996, ChartObj.APRIL,  5),
                           new ChartCalendar(1996, ChartObj.MAY,  5),
                           new ChartCalendar(1996, ChartObj.JUNE,  5),
                           new ChartCalendar(1996, ChartObj.JULY,  5),
                           new ChartCalendar(1996, ChartObj.AUGUST,  5),
                           new ChartCalendar(1996, ChartObj.SEPTEMBER, 5),
                           new ChartCalendar(1996, ChartObj.OCTOBER, 5),
                           new ChartCalendar(1996, ChartObj.NOVEMBER,  5)};
double [] yData = {10, 22, 33, 44, 55, 46, 33, 25, 14, 9};

TimeSimpleDataset dataset =  new TimeSimpleDataset("Sales", xData, yData);
TimeCoordinates simpleTimeScale = new TimeCoordinates();
simpleTimeScale.AutoScale(dataset);
```

Using similar logic you can define a **TimeSimpleDataset** with numeric x-values and **ChartCalendar** y-values. If you auto-scale the coordinate system using that dataset you will end up with a numeric x-axis and a time y-axis.

You can control the "tightness" of the auto-scale values about the dataset values using other versions of the **TimeCoordinates.AutoScale** method that take rounding mode parameters.

Example of auto-scaling a TimeCoordinates object using a single dataset and explicit rounding mode parameters

```
simpleTimeScale.AutoScale(dataset,ChartObj.AUTOAXES_FAR, ChartObj.AUTOAXES_FAR)
```

You can auto-scale the coordinate bounds using a dataset, and then explicitly modify the range the auto-scale selected. There are methods for setting the minimum and maximum values of the x- and y-scales. This way you can use the auto-scale methods for the values of one scale (the y-scale in the example below), but explicitly set the values for the other scale (the x-scale in the example below).

Example of modifying the minimum and maximum values selected by an auto-scale method

[C#]

```
ChartCalendar [] xData = { new ChartCalendar(1996, ChartObj.FEBRUARY,  5),
                           new ChartCalendar(1996, ChartObj.MARCH,  5),
                           new ChartCalendar(1996, ChartObj.APRIL,  5),
                           new ChartCalendar(1996, ChartObj.MAY,  5),
                           new ChartCalendar(1996, ChartObj.JUNE,  5),
                           new ChartCalendar(1996, ChartObj.JULY,  5),
                           new ChartCalendar(1996, ChartObj.AUGUST,  5),
                           new ChartCalendar(1996, ChartObj.SEPTEMBER, 5),
                           new ChartCalendar(1996, ChartObj.OCTOBER, 5),
                           new ChartCalendar(1996, ChartObj.NOVEMBER,  5)};
double [] yData = {10, 22, 33, 44, 55, 46, 33, 25, 14, 9};

TimeSimpleDataset dataset =  new TimeSimpleDataset("Sales", xData, yData);
TimeCoordinates simpleTimeScale = new TimeCoordinates();
simpleTimeScale.AutoScale(dataset);
simpleTimeScale.SetTimeScaleStart(new ChartCalendar(1996, ChartObj.JANUARY,5));
simpleTimeScale.SetTimeScaleStop(new ChartCalendar(1997, ChartObj.JANUARY,5));
```

The auto-scale methods that use an array of datasets to determine the proper range are very similar.

Example of auto-scaling a TimeCoordinates object using the multiple datasets

[C#]

```
ChartCalendar [] xData = { new ChartCalendar(1996, ChartObj.FEBRUARY,  5),
                           new ChartCalendar(1996, ChartObj.MARCH,  5),
                           new ChartCalendar(1996, ChartObj.APRIL,  5),
                           new ChartCalendar(1996, ChartObj.MAY,  5),
                           new ChartCalendar(1996, ChartObj.JUNE,  5),
                           new ChartCalendar(1996, ChartObj.JULY,  5),
                           new ChartCalendar(1996, ChartObj.AUGUST,  5),
                           new ChartCalendar(1996, ChartObj.SEPTEMBER, 5),
                           new ChartCalendar(1996, ChartObj.OCTOBER, 5),
                           new ChartCalendar(1996, ChartObj.NOVEMBER,  5)};

double [] yData1 = {10, 22, 33, 44, 55, 46, 33, 25, 14, 9};
double [] yData2 = {20, 12, 43, 54, 15, 26, 63, 25, 24, 19};
double [] yData3 = {30, 52, 13, 64, 25, 76, 13, 35, 24, 19};
// All of the datasets reference the same xData array of ChartCalendar
// dates, though this does not have to be the case.
TimeSimpleDataset dataset1 = new TimeSimpleDataset("Sales1",xData,yData1);
```

```
TimeSimpleDataset dataset2 = new TimeSimpleDataset("Sales2",xData,yData2);
TimeSimpleDataset dataset3 = new TimeSimpleDataset("Sales3",xData,yData3);
TimeSimpleDataset [] datasetsArray = new TimeSimpleDataset[3];
datasetsArray[0] = dataset1;
datasetsArray[1] = dataset2;
datasetsArray[2] = dataset3;
TimeCoordinates simpleTimeScale = new TimeCoordinates();
simpleTimeScale.AutoScale(datasetsArray);
```

There is a version of the multiple dataset auto-scale routine that also specifies rounding mode parameters.

```
simpleTimeScale.AutoScale(datasetsArray,ChartObj.AUTOAXES_FAR, ChartObj.AUTOAXES_FAR);
```

The previous examples use the **TimeCoordinates** default 7 days/week, and 24-hours/day modes. Many of the methods in the software have a *weektype* parameter. The default value of this parameter is the constant **ChartObj.WEEK_7D**. If you want the coordinate system to ignore Saturdays and Sundays, use the constant **ChartObj.WEEK_5D** constant**.** Use the methods below to establish a 5-days/week coordinate system.

TimeCoordinates constructor

```
[C#]
public TimeCoordinates(
   ChartCalendar dstart,
   double y1,
   ChartCalendar dstop,
   double y2,
   int nweektype
);
public TimeCoordinates(
  double x1,
  ChartCalendar dstart,
  double x2,
  ChartCalendar dstop,
);
public TimeCoordinates(
   ChartCalendar dstart,
   double y1,
   ChartCalendar dstop,
   double y2,
);
```

SetTimeCoordinateBounds method

```
[C#]
public void SetTimeCoordinateBounds(
   ChartCalendar dstart,
   double y1,
   ChartCalendar dstop,
   double y2,
   int nweektype
);
```

```
public void SetTimeCoordinateBounds(
   ChartCalendar dstart,
   double y1,
   ChartCalendar dstop,
   double y2,
);

public void SetTimeCoordinateBounds(
   double x1,
   ChartCalendar dstart,
   double x2,
   ChartCalendar dstop,
);
```

SetWeekType method

```
[C#]
public void SetWeekType(
   int weektype
);
```

If you use the auto-scale routines, set the week type before you call the **TimeCoordinates.AutoScale** method because the auto-scale routines need to take into account the week type. For example:

[C#]

```
TimeSimpleDataset dataset =  new TimeSimpleDataset("Sales", xData, yData);
TimeCoordinates simpleTimeScale = new TimeCoordinates();
simpleTimeScale.SetWeekType(ChartObj.WEEK_5D);
simpleTimeScale.AutoScale(dataset);
```

In addition to the week type, the other major way to customize a **TimeCoordinates** coordinate system is not to use a 24-hour day. There are methods that set the starting and ending time-of-day. For example, if you are interested in plotting stock market data trading during the regular trading day of 9:30 AM to 4:00 PM, you can setup the coordinate system to only include these hours, and to treat any data outside of these hours as invalid and not to be plotted. A day can have only one continuous range. You are not able to define a day to with a valid range of 9:30 AM to 4:00 PM and 6:00 PM to 9:00 PM. Only one of these ranges is valid, or a combined range of 9:30 AM to 9:00 PM where the 4:00 PM to 6:00 PM time segment is included in the range.

TimeCoordinates constructor with time-of-day parameters

```
[C#]
public TimeCoordinates(
   ChartCalendar dstart,
   long starttime,
   double y1,
   ChartCalendar dstop,
   long stoptime,
   double y2,
   int nweektype
);
```

```
public TimeCoordinates(
   ChartCalendar dstart,
   long starttime,
   double y1,
   ChartCalendar dstop,
   long stoptime,
   double y2,
   int ntimeaxis
   int nweektype
);
```

TimeCoordinates constructor example for a 5 day/week and 9:30 AM to 4:00 PM time-of-day range

[C#]

```
long starttod = (9 * 60 + 30) * 60 * 1000; // msecs cooresponding to 9:30 AM
long stoptod = 16 * 60 * 60 * 1000; // msecs cooresponding to 4:00 PM
ChartCalendar dstart = new ChartCalendar(1996,ChartObj.FEBRUARY,5);
ChartCalendar dstop = new ChartCalendar(1997, ChartObj.JANUARY,5);
double y1 = 0.0;
double y2 = 55.0;
TimeCoordinates stockTimeScale;

stockTimeScale =   new TimeCoordinates(dstart, starttod,  y1, dstop, stoptod , y2,
                                       ChartObj.WEEK_5D)
```

Another technique uses the default constructor and scales the coordinates using the **TimeCoordinates.SetTimeCoordinateBounds** method.

SetTimeCoordinateBounds Method

```
[C#]
public void SetTimeCoordinateBounds(
   ChartCalendar dstart,
   long starttod,
   double y1,
   ChartCalendar dstop,
   long stoptod,
   double y2,
   int nweektype
);

public void SetTimeCoordinateBounds(
   double x1,
   ChartCalendar dstart,
   long starttod,
   double x2,
   ChartCalendar dstop,
   long stoptod,
   int nweektype
);
```

SetTimeCoordinateBounds example for a 5 day/week and 9:30 AM to 4:00 PM time-of-day range SetWeekType method

[C#]

```
long starttod = (9 * 60 + 30) * 60 * 1000; // msecs cooresponding to 9:30 AM
long stoptod = 16 * 60 * 60 * 1000; // msecs cooresponding to 4:00 PM
```

```
ChartCalendar dstart = new ChartCalendar(1996, ChartObj.FEBRUARY,  5);
ChartCalendar dstop = new ChartCalendar(1997, ChartObj.JANUARY,  5);
double y1 = 0.0;
double y2 = 55.0;
TimeCoordinates stockTimeScale;

stockTimeScale = new TimeCoordinates();
stockTimeScale.SetTimeCoordinateBounds(dstart, starttod,  y1,
                                       dstop, stoptod , y2,
                                      ChartObj.WEEK_5D);



                                  dstop, stoptod, y2, ChartObj.WEEK_5D)
```

If you use the auto-scale routines, set the week type and the time-of-day range before you call the **TimeCoordinates.AutoScale** method because the auto-scale routines need to take into account the number of seconds per day and the week type. For example:

[C#]

```
// the tradingDay array is initialized with the stock trading dates
// the stockPrice array is initialized with stock price data
TimeSimpleDataset Dataset1 = new TimeSimpleDataset("First",tradingDay,stockPrice);

long startTime = (9 * 60 + 30) * 60 * 1000; // msecs cooresponding to 9:30 AM
long stopTime = 16 * 60 * 60 * 1000; // msecs cooresponding to 4:00 PM

TimeCoordinates stockTimeScale = new TimeCoordinates();
stockTimeScale.SetWeekType (ChartObj.WEEK_5D);
stockTimeScale.SetScaleStartTOD(startTime);
stockTimeScale.SetScaleStopTOD(stopTime);
stockTimeScale.AutoScale(Dataset1,ChartObj.AUTOAXES_FAR, ChartObj.AUTOAXES_FAR);
```

Class ElapsedTimeCoordinates

PhysicalCoordinates

|

+-- CartesianCoordinates

|

+-- ElapsedTimeCoordinates

The **ElapsedTimeCoordinates** class scales the chart plot area for a physical coordinate system which uses an elapsed time scale in combination with a linear or logarithmic scaling. The elapsed time scale uses milliseconds as the time base, so all time values should be represented using their milliseconds equivalent value, i.e. a value of 30 seconds is represented by the value 30000. The axis labeling class, **ElapsedTimeAxisLabels**, converts the millisecond values to equivalent seconds values, i.e., the value 30000 milliseconds will be displayed as 30 seconds in the format "00:00:30".

There are three main ways to scale the plot area:

- Scale the minimum and maximum x- and y- values explicitly. You can scale the the elapse time axis using **TimeSpan** objects, or using millisecond values, i.e. an elapsed time range of 0 – 30 seconds would be scaled for 0-30000.

- Use an auto-scale method to calculates appropriate minimum and maximum x- and y-values based on the x- and y-values in one or more datasets
- Use a combination of the first two methods. It is useful to be able to run an auto-scale function, and then change the minimum or maximum value of one or more coordinate endpoints.

ElapsedTime Coordinate Scaling

The default coordinate system for the **ElapsedTimeCoordinates** class is elapsed time for the x-dimension and linear for the y dimension. If you already know the range for x and y for the plot area, you can scale the plot area explicitly.

The example below uses a **ElapsedTimeCoordinates** constructor to initialize the coordinates to the proper values.

Scale for elapsed time using the ElapsedTimeCoordinates constructor with explicit scaling for a range of 30 seconds.

[C#]

```
double xMin = 0; // starting elapsed time is 0
double xMax = 30 * 1000; // ending elpase time is 30 seconds
double yMin = 0;
double yMax =  105;

ElapsedTimeCoordinates simpleScale;
simpleScale = new ElapsedTimeCoordinates(xMin, yMin, xMax, yMax);

TimeSpan xTSMin = TimeSpan.FromSeconds(0); // starting elapsed time is 0
TimeSpan xTSMax = TimeSpan.FromSeconds(30); // ending elpase time is 30 seconds
simpleScale = new ElapsedTimeCoordinates(xTSMin, yMin, xTSMax, yMax);
```

Another technique uses the default constructor and scales the coordinates using the **ElapsedTimeCoordinates .SetCoordinateBounds** method.

Example of explicit scaling of a ElapsedTimeCoordinates object using the ElapsedTimeCoordinates.SetCoordinateBounds method

[C#]

```
double xMin  = 0  ' starting elapsed time is 0
double xMax = 30 * 1000 ' ending elpase time is 30 seconds
double yMin = 0;
double yMax =  105;
ElapsedTimeCoordinates simpleScale = new ElapsedTimeCoordinates ();

simpleScale.SetCoordinateBounds(xMin, yMin, xMax, yMax);
```

It is possible to scale the bounds of the coordinate system based on the data values in a dataset. There are constructors and methods that take a single dataset and others that take an array of datasets.

Example of auto-scaling a ElapsedTimeCoordinates object using a single dataset

[C#]

```
double [] xData = {1000,2000,3000,4000,5000,6000,7000,8000,9000,10000};
double [] yData = {10, 22, 33, 44, 55, 46, 33, 25, 14, 9};

ElapsedTimeSimpleDataset dataset = new ElapsedTimeSimpleDataset ("Sales", xData, yData);
ElapsedTimeCoordinates simpleScale  = new ElapsedTimeCoordinates ();
simpleScale.AutoScale(dataset);
```

You can control the "tightness" of the auto-scale values about the dataset values using other versions of the **ElapsedTimeCoordinates .AutoScale** method that take rounding mode parameters.

Example of auto-scaling a ElapsedTimeCoordinates object using a single dataset and explicit rounding mode parameters

```
simpleScale.AutoScale(dataset,  ChartObj.AUTOAXES_FAR, ChartObj.AUTOAXES_FAR)
```

You can auto-scale the bounds of the coordinate system using a dataset, and then explicitly modify the range the auto-scale selected. There are methods that set the minimum and maximum values of the x- and y-scales. This way you can use the auto-scale methods for the values of one scale (the y-scale in the example below), but explicitly set the values for the other scale (the x-scale in the example below).

Example of modifying the minimum and maximum values selected by an auto-scale method.

[C#]

```
double [] xData = {1000,2000,3000,4000,5000,6000,7000,8000,9000};
double [] yData = {11, 22, 33, 44, 55, 46, 33, 25, 14};

ElapsedTimeSimpleDataset dataset = new ElapsedTimeSimpleDataset ("Sales", xData, yData);
ElapsedTimeCoordinates simpleScale  = new ElapsedTimeCoordinates ();
simpleScale.AutoScale(dataset);
simpleScale.SetScaleStartY(0);
simpleScale.SetScaleStopY(100.0);
```

The auto-scale methods that use an array of datasets to determine the proper range are very similar.

Example of auto-scaling a ElapsedTimeCoordinates object using the multiple datasets

[C#]

```
double [] xData1 = {1000,2000,3000,4000,5000,6000,7000,8000,9000,10000};
double [] yData1 = {10, 22, 33, 44, 55, 46, 33, 25, 14, 9};
double [] xData2 = 1000,2000,3000,4000,5000,6000,7000,8000,9000,10000};
double [] yData2 = {20, 12, 43, 54, 15, 26, 63, 25, 24, 19};
double [] xData3 = 1000,2000,3000,4000,5000,6000,7000,8000,9000,10000};
double [] yData3 = {30, 52, 13, 64, 25, 76, 13, 35, 24, 19};

ElapsedTimeSimpleDataset dataset1 = new ElapsedTimeSimpleDataset ("Sales1",xData1,yData1);
ElapsedTimeSimpleDataset dataset2 = new ElapsedTimeSimpleDataset ("Sales2",xData2,yData2);
ElapsedTimeSimpleDataset dataset3 = new ElapsedTimeSimpleDataset ("Sales3",xData3,yData3);
ElapsedTimeSimpleDataset [] datasetsArray = new ElapsedTimeSimpleDataset [3];
datasetsArray[0] = dataset1;
datasetsArray[1] = dataset2;
datasetsArray[2] = dataset3;
ElapsedTimeCoordinates simpleScale  = new ElapsedTimeCoordinates ();
simpleScale.AutoScale(datasetsArray);
```

There is a version of the multiple dataset auto-scale routine that also specifies rounding mode parameters.

```
simpleScale.AutoScale(datasetsArray,ChartObj.AUTOAXES_FAR, ChartObj.AUTOAXES_FAR)
```

Class EventCoordinates

PhysicalCoordinates

|

+-- CartesianCoordinates

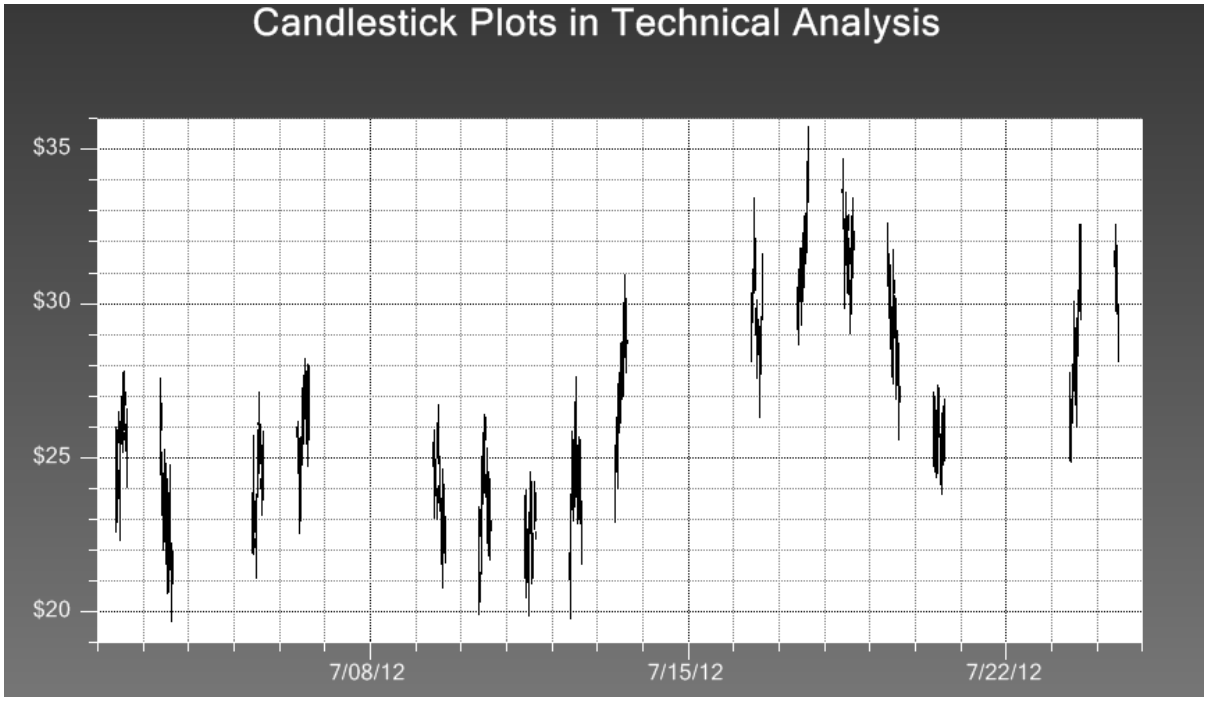
|

+-- EventCoordinates

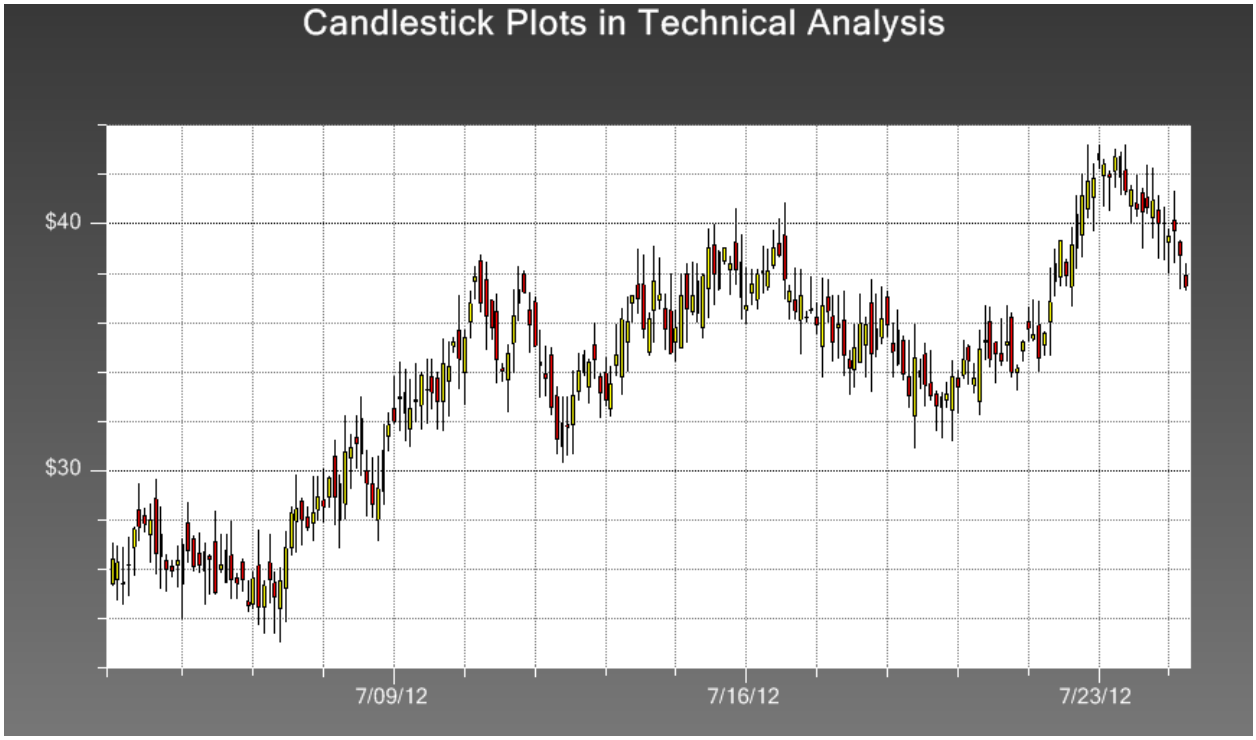
The **EventCoordinates** class scales the chart plot area for a physical coordinate system which uses an event scale in combination with a linear or logarithmic scaling. The underlying event scale uses a simple linear scale, scaled from 0 to N-1, where N is the number of ChartEvents with a unique time stamp in the attached ChartEvent based datasets: EventSimpleDataset and EventGroupDataset. Unlike the other coordinate systems, an EventCoordinate object requires an event dataset as part of its definition.

The **ChartEvent** class incorporates two x-value positioning properties, the Position and the TimeStamp, and one or more numeric y-values for each event. A single event therefore defines both the x-and y-values of the event in the underlying coordinate system. A collection, or array, of ChartEvent objects define the data for a plot, the same way as arrays of x- and y-values define a plot when using a simple dataset class with a Cartesian coordinate system. The critical element of the ChartEvent which permit it to be used for the plotting of discontinuous data is that the Position of the event in a chart is related, but, independent of the TimeStamp of the event. Event data can be positioned contiguously, and evenly spaced, in a chart, even if the time stamps of the

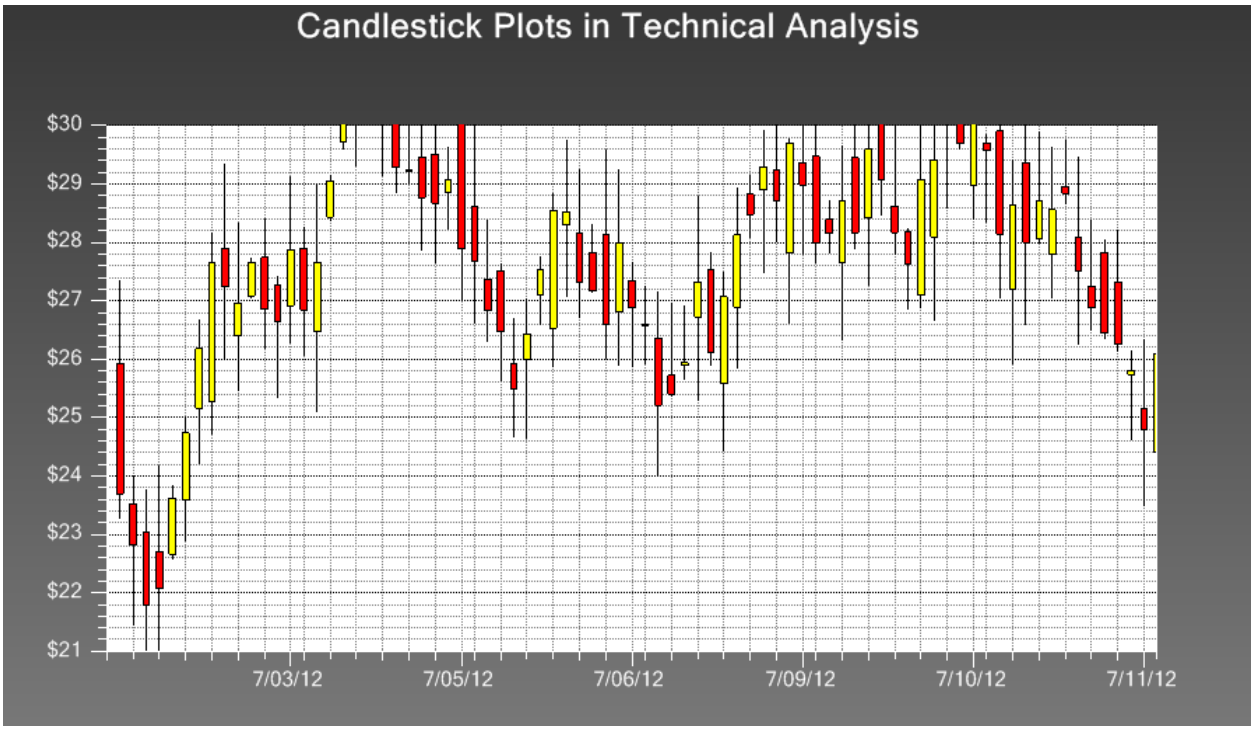
events are not contiguous, or evenly spaced. Here is a simple example of a standard financial candlestick plot chart, using our TimeCoordinates class as the coordinate system, where the time/date data is not evenly spaced, and contains large gaps corresponding to weekends, and inactive hours of the day. The July 4th holiday is included in the range, and there is no data for that time interval either.



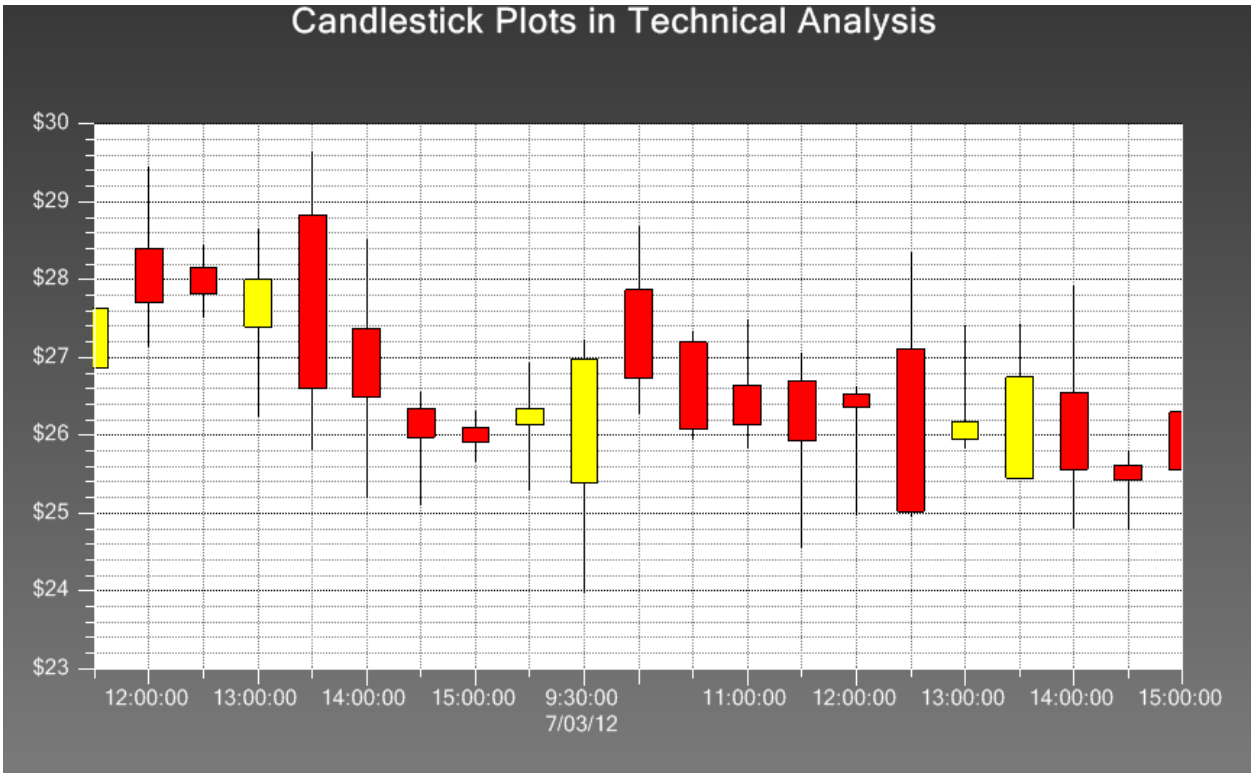
Contrast this to the similar data, using the same time range, plotted using the EventCoordinates class. Note how every event is evenly spaced with its neighbor. Gaps do not exist, since weekends, holidays, and unused hours are bridged over as if they do not exist.he same would be true for gaps due to holidays, and a varying number of work hours in a day.



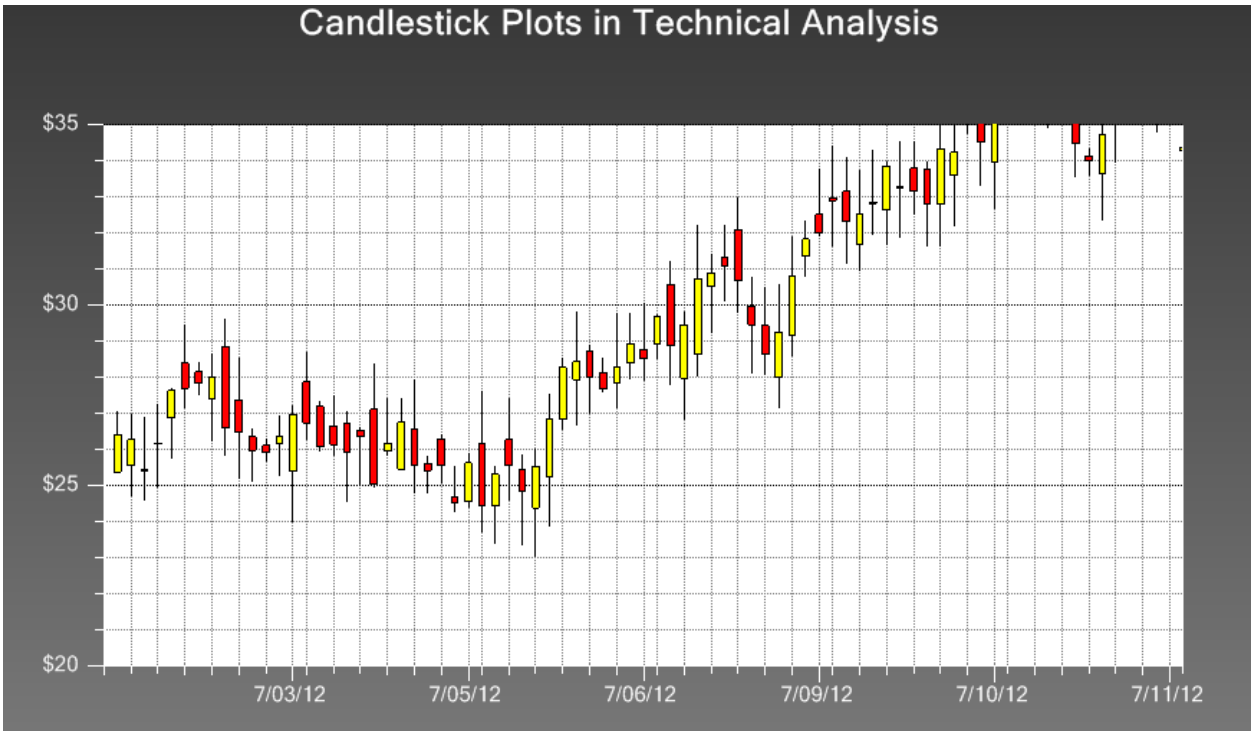
Zooming in further, you can see the smooth transition across the July 4th holiday, and the following weekend.



Zoom in again, and you can see the smooth transition from one day to the next, even though the working hours are only a 9:30 to 16:00 subset of the 24 hours of a day.



This is accomplished because of the dual positioning values, Position and TimeStamp, of the ChartEvent class. Each element of a plot object (one of the candlestick objects in the plot above) is positioned in a simple linear coordinate system, starting at 0 and incrementing by 1 for each ChartEvent object. In the previous example, the first ChartEvent object is has a Position value of 0.0, and the last ChartEvent object has a position of 199, because there are 200 data points in the chart. This is what keeps the individual elements of a plot object evenly spaced, because the plot elements are positioned in the chart using the Position value, not the TimeStamp value. But, the associated x-axis (EventAxis) and x-axis labels objects (EventAxisLabels) look to the TimeStamp property for their values, not the Position property. What you end up with is the clean, evenly spaced look of a simple linear chart, with the axis tick marks and axis labeling of a dedicated time/date axis. The graph can be made to scroll (or pan) left to right, or re-scale along the y-axis, smoothly.



Creating a chart using the event classes uses the same basic sequence as our other coordinate systems.

First, create the data – in this case an array of ChartEvent objects. Most of the code below is just the simulation of some raw data, taking into account a 9:30 to 16:00 working day, with no weekends. Note that each ChartEvent object is defined using both a TimeStamp value (xvalues[i]), and a Position value (i). The value of i controls the position of the ChartEvent object in the plot, and the value of the TimeStamp controls the x-axis tick marks and labels.

[C#]

```
double minval = 0.0, maxval = 0.0;
int incrementbase = ChartObj.MINUTE;
int increment = 10;
ChartCalendar currentdate = new ChartCalendar();
ChartEvent[] eventArray = new ChartEvent[nNumPnts];
stockPriceData[3] = 25.5;  // close
stockPriceData[0] = 24.5;   // open
stockPriceData[1] = 26;   // high
stockPriceData[2] = 24;   // low

ChartEvent currentEvent = new ChartEvent();
currentdate = ChartCalendar.CalendarDaysAdd(currentdate, 1, weekmode);
currentdate.SetTOD(9, 33, 0);
for (i = 0; i < nNumPnts; i++)
{
    double position = i + 1;
    xValues[i] = (ChartCalendar)currentdate.Clone();
    if (i > 0)
    {
        stockPriceData[3] += 2 * (0.5 - ChartSupport.GetRandomDouble()); // close
        stockPriceData[0] += 2 * (0.5 - ChartSupport.GetRandomDouble()); // open
```

```
        minval = Math.Min(stockPriceData[3], stockPriceData[0]);
        maxval = Math.Max(stockPriceData[3], stockPriceData[0]);
       stockPriceData[1] = maxval + 1.5 * ChartSupport.GetRandomDouble();  // high
        stockPriceData[2] = minval - 1.5 * ChartSupport.GetRandomDouble();  // low
    }

    currentEvent = new ChartEvent(xValues[i], position, stockPriceData);
    currentEvent.AxisLabel = "XXX" + position.ToString();
    currentEvent.ToolTip = "ToolTip" + position.ToString();
    eventArray[i] = currentEvent;

    currentdate.Add(incrementbase, increment);
    if (currentdate.Get(ChartObj.HOUR_OF_DAY) >= 16)
    {
        currentdate.Add(ChartObj.DAY_OF_YEAR, 1);
        currentdate.SetTOD(9, 30, 0);
    }
}
```

Create an EventSimpleDataset, or EventGroupDataset using the source data.

[C#]

```
EventGroupDataset Dataset1 = new EventGroupDataset("Stock Data", eventArray, 4);
```

Create an EventCoordinateSystem, referencing the EventSimpleDataset as a parameter. The coordinate system is defined by the content of the EventSimpleDataset. It will auto-scale the coordinate system to the number of ChartEvents found in the source dataset.

[C#]

```
pTransform1.AutoScale(Dataset1,ChartObj.AUTOAXES_NEAR,  ChartObj.AUTOAXES_NEAR);
```

Define the x-axis as an EventAxis. The tick marks of the axis are defined using the TickRule property, in this case the TickRule is TICK_RULE.MINORCROSSOVEREVENT_MAJORCROSSOVEREVENT. This means a minor tick mark is placed every time the time rolls over a minor event, and a major tick mark is placed every time the time rolls over a major event. More on this later.

[C#]

```
EventAxis xAxis1 = new EventAxis(pTransform1);
xAxis1.SetColor(Colors.White);
xAxis1.TickRule = ChartObj.TICK_RULE.MINORCROSSOVEREVENT_MAJORCROSSOVEREVENT;
chartVu.AddChartObject(xAxis1);
```

Last, define the x-axis labels using the EventAxisLabels class.

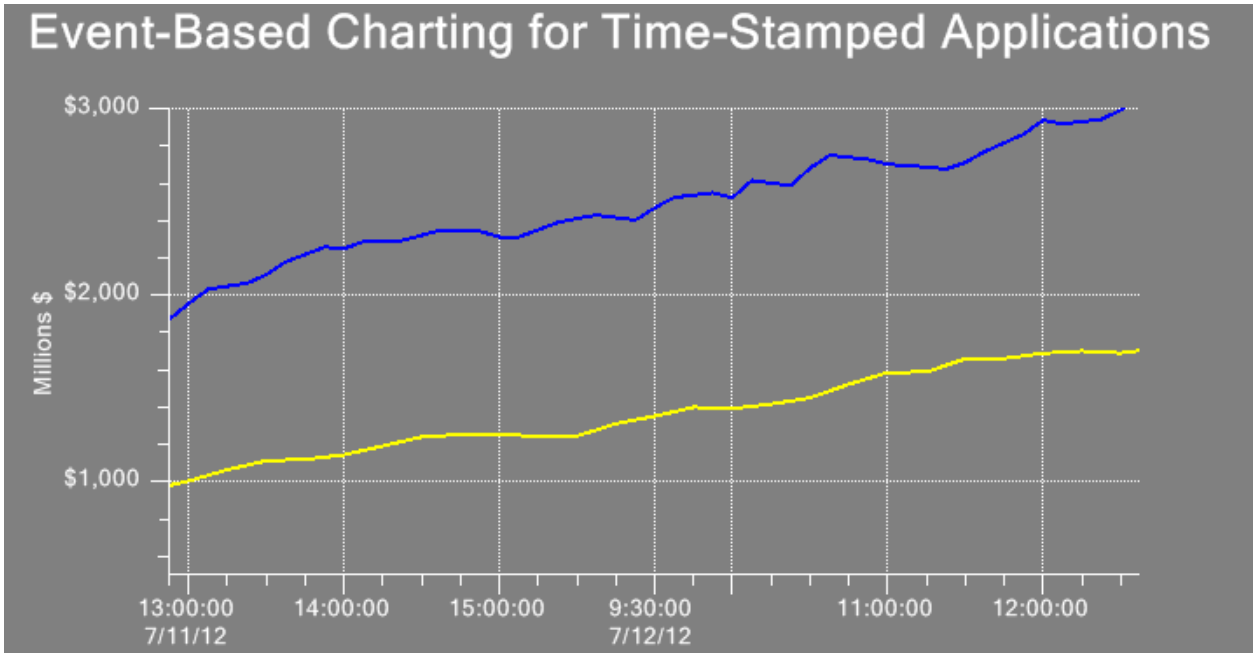
[C#]

```
EventAxisLabels xAxisLab1 = new EventAxisLabels(xAxis1);
xAxisLab1.SetColor(Colors.White);
chartVu.AddChartObject(xAxisLab1);
```

Everything else is the same as in our other charts.

Things are more complicated once you start plotting multiple, overlapping, datasets in the same coordinate system. Each ChartEvent object in the previous example had a unique Position and TimeStamp value. The software only needed to auto-scale the coordinate system for the range of ChartEvent Position values, and plot the data. If you want to plot a second dataset in the same chart, it is more complicated. If every ChartEvent object of the the second dataset uses exactly the same Position and TimeStamp values as the first, and only has different y-values, you could plot it as is. You should be able to plot the second dataset directly on top of the first, and the TimeStamps of the ChartEvents objects would line up chronologically exactly as you would expect. The complication arises if the second dataset overlaps the first with respect to the TimeStamp values, but does NOT use exactly the same TimeStamps as the first set of data. For example, the first set of data is contains ChartEvent objects sampled at 10 minute intervals, starting at 8:30. The second dataset contains ChartEvents objects starting at 8:15 and sampled every 15 minutes. In this case some of the TimeStamp values would match (at every ½ hour), but in every case, the Position values would be out of sync. If these two datasets were plotted as is, the TimeStamp values for the two datasets would not align with respect to the x-axis.

When multiple datasets are attached to the same EventCoordinate system, the coordinate system needs to merge and sort the ChartEvent objects of every dataset. The ChartEvents are sorted by the TimeStamp value. It may be that many TimeStamps are duplicates, since different datasets may have used the same TimeStamps in their event data. Other TimeStamps may be singular, having occurred in only one dataset. It really doesn't matter. Next the software runs through the ChartEvents of the sorted list, assigning a new Position value to every object in the list. If adjacent ChartEvent objects in the sorted list have the same TimeStamp, they are assigned the same Position value. If the next object in the sorted list has a different TimeStamp value, it is assigned a Position value one greater than the previous ChartEvent. The net effect is that the ChartEvents with the same TimeStamp value will align properly at the same x-position in the graph, regardless of how the initial Position values were set. The EventAxis and EventAxisLabels objects, when applied to the EventCoordinate system, will see every unique TimeStamp in the merged datasets, without double counting duplicate TimeStamp values across dataset.



Note how the plot still captures the crossover between 16:00 7/12/12 and 9:30 7/13/12 without a gap. The same would be true if there was a weekend, or holiday, or even a lunch break, between adjacent data points.

Below is an example of how multiple datasets are attached to an EventCoordinates system.

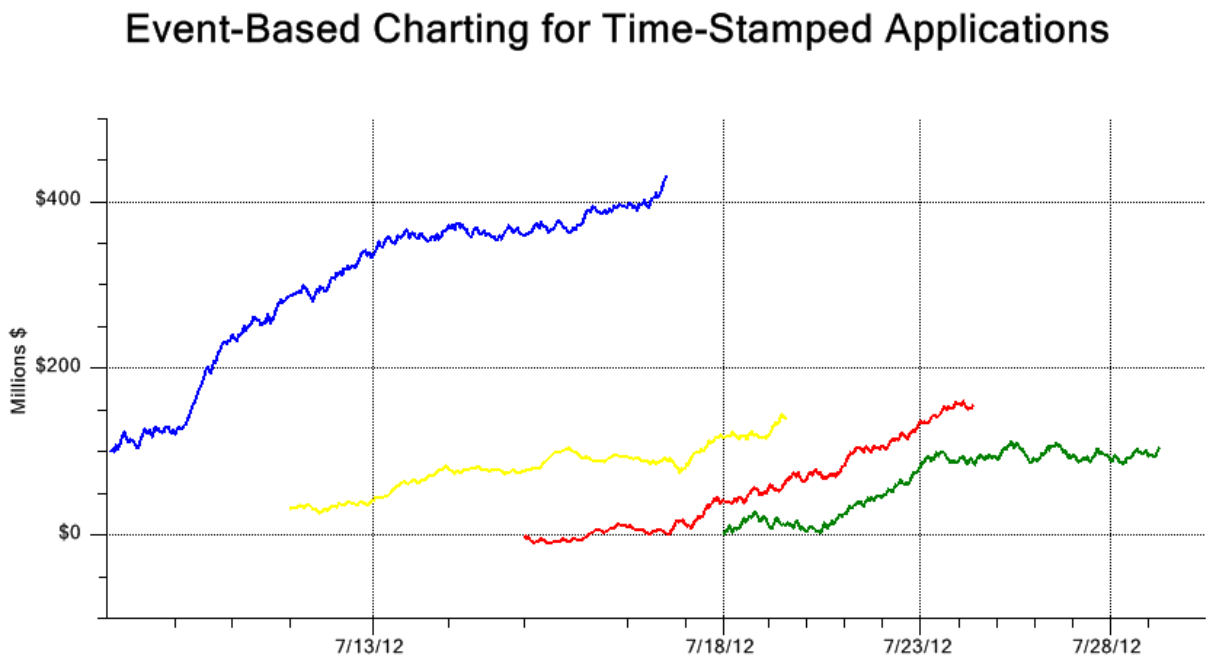
[C#]

```
EventSimpleDataset Dataset1 = new EventSimpleDataset("Actual Sales", cev1);
EventSimpleDataset Dataset2 = new EventSimpleDataset("Forecast Sales", cev2);
EventSimpleDataset Dataset3= new EventSimpleDataset("Actual Sales", cev3);
EventSimpleDataset Dataset4 = new EventSimpleDataset("Forecast Sales", cev4);

EventSimpleDataset[] DatasetArray = { Dataset1, Dataset2, Dataset3, Dataset4 };

EventCoordinates pTransform1 = new EventCoordinates(DatasetArray);
```

This method relies on the ability to detect when the time stamps of an event are equal. In the case of pure time, equality depends on the granularity you want in the display. Events can be equal at the year, month, week, day, hour, minute second and millisecond level. So, if you want events, across multiple plots, to line up by the minute, not caring for differences in seconds or milliseconds, you can do that. Set the EventCoordinates property TimeStampResolution to ChartObj.MINUTE. The default value is ChartObj.SECOND, and if you want you can set the resolution to MILLISECOND, SECOND, MINUTE, HOUR, DAY_OF_YEAR, WEEK_OF_YEAR, MONTH, or YEAR. Make sure you set the resolution to a value below that what you want to see in your data. If your data is sampled at 6 second intervals, and you want to

see each value at a unique position on the x-axis, set the TimeStampResolution to SECOND. If you set it to MINUTE, all of the samples within a minute interval will be grouped together at a single Position value.

Below is an example of a chart with multiple, overlapping datasets, sampled at different resolutions.



When the merged datasets have the Position values of their ChartEvent objects modified to repflect the true position of the ChartEvent in the graph, the auto-positioning starts at 1 (not 0). If 0 was used, the first data point would exactly on the edge of the clipping window, and this would cut off half of a bar, scatter plot, candlestick, or OHLC symbol positioned in the first position. The default auto-scaling values scale the ChartCoordinates scale from 0 to N+1, where N is then number of unique ChartEvent Positon values. This way ChartEvent objects are not cut off on the left or the right.

There is a specialized axis class, EventAxis, and axis labels class, EventAxisLabels, to use with event data. Unlike our regular LinearAxis, and TimeAxis classes, which are independent of the data in the chart, the EventAxis is dependent on the underlying data. The tick marks of the EventAxis are placed at the x-position of the associated ChartEvent objects. If there are more than 10-20 ChartEvent objects in the graph, the tick mark labels would start to overlap, so we divide the tick marks into major tick marks, which are those that get a label, and minor tick marks which do not get a label. Further, if there are more than 100-200 ChartEvent objects in the graph, then the tick marks may start to overlap. So the software has the option of drawing a minor, or a major tick mark, every Nth event, to keep them from overlapping. Most of this is taken care of by the auto-axis routines. Though it is possible you do not like the results – you

can't please everybody. So, once the axis is created you can modify the appearance by adjusting the following properties.

Regardless of how you initially setup the graph, if you use the zoom routines, or the RTScrollFrame routines in the QCRTGraph software, the look of the axes will always revert to our auto-scaling, not your modified setup. Because we cannot predict what you will do, and cannot scale a completely different range of values based on whatever logic you use.

TickRule – Controls the tick mark logic of the axis. User one of the TICK_RULE enumeration constants:

NO_TICKS – do not display any tick marks. No tick marks means no axis labels.

MINOREVENT_MAJOREVENT – display a minor tick mark every AxisMinorNthTick event, and a major tick mark every AxisMinorTicksPerMajor.

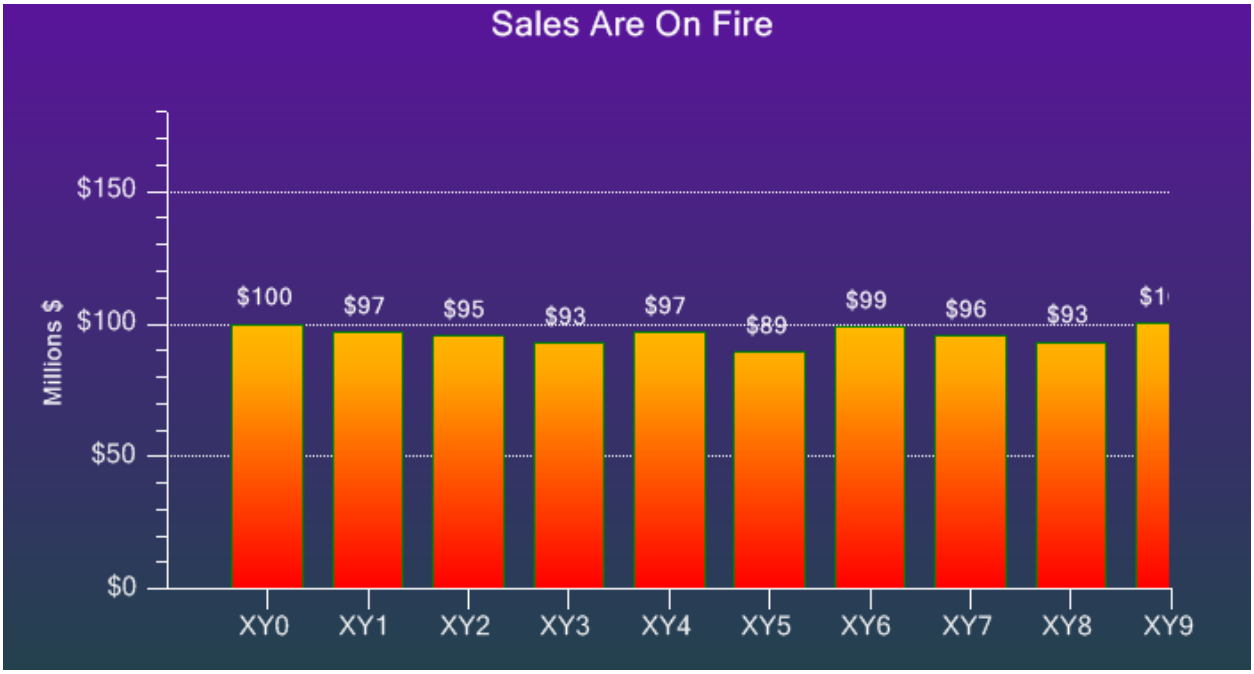
MAJOREVENT - display a major tick mark every AxisMajorNthTick event.

MINORCROSSOVEREVENT_MAJORCROSSOVEREVENT – display a minor tick mark every AxisMinorNthTick, minor crossover event, and a major tick mark every AxisMajorNthTick major crossover event. The minor and major crossover events are controlled by the MajorTickCrossoverEvent and MinorTickCrossoverEvent properties of the EventAxis.

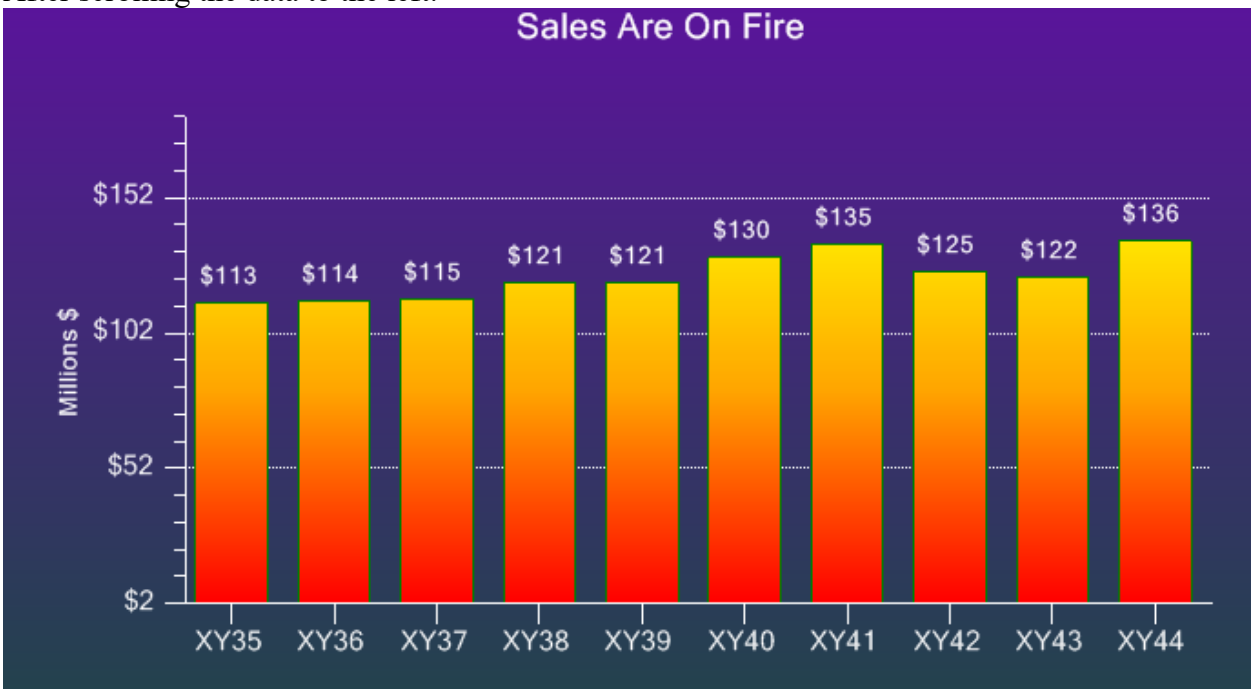
The default is
ChartObj.TICK_RULE.MINORCROSSOVEREVENT_MAJORCROSSOVEREVENT.

The term crossover event means that a field of date/time timestamp changes. If you specify a MinorTickCrossoverEvent of ChartObj.SECOND, and an AxisMinorNthTick of 15, this will cause a minor tick mark to be displayed every 15th second, if an event falls within that range. So, if your events are spaced approximately 5 seconds apart, you will get a minor tick mark for approximately every three events. If you choose a MajorTickCrossoverEvent of ChartObj.MINUTE and an AxisMajorNthTick of 1, this will cause a major tick mark to be displayed every minute, if an event falls within that range. Tick marks only show up on an event, so if there are no events within the time interval, no tick mark will appear.

Every tick mark can have a custom label. So if you do not want to use the default time/date labels, but instead want label the event tick marks with a custom string, you can do that. The string will track the exact tick mark associated with a given event. This makes scrolling through the data easier, because the custom tick mark strings stick to the tick mark, and don't have to be recalculated.



After scrolling the data to the left:

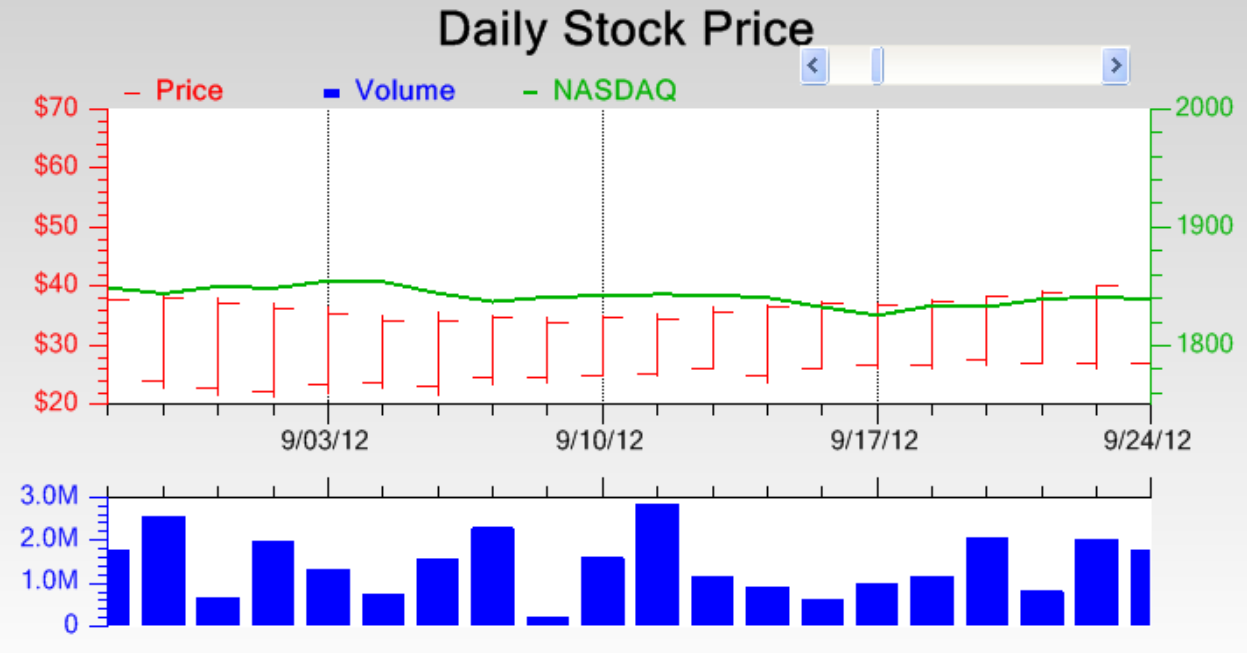


If you plan to implement scrolling (panning) along the x-axis, using a scroll bar, or some other method, you need to know how to re-scale the x-scale EventCoordinate system. First,

understand that the underlying coordinate system is a Cartesian coordinate system, with the x-axis scaled from 0 to the number of ChartEvent objects with unique time stamps. Because a simple linear scale is used for the x-axis, you can scale the x-axis using simple linear values (0 to number of ChartEvent objects). In that case you use the EventCoordinates.ScaleStartX and EventCoordinates.ScaleStopX properties. The code below is extracted from the **ChartEventExamples.OHLCEventChart** example program.

[C#]

```
public void UpdateScaleAndAxesUsingEventIndex(int startindex)
 {

     pTransform1.ScaleStartX = startindex;
     pTransform1.ScaleStopX = startindex + numberEventsInView - 1;

     pTransform2.ScaleStartX = startindex;
     pTransform2.ScaleStopX = startindex + numberEventsInView - 1;

     pTransform3.ScaleStartX = startindex;
     pTransform3.ScaleStopX = startindex + numberEventsInView - 1;


     xAxis1.CalcAutoAxis();
     yAxis1.CalcAutoAxis();
     xAxisLab1.CalcAutoAxisLabels();
     yAxisLab1.CalcAutoAxisLabels();

     xAxis2.CalcAutoAxis();
     xAxis2.SetAxisIntercept(pTransform2.GetStopY());
     xAxis2.SetAxisTickDir(ChartObj.AXIS_MAX);

     yAxis2.CalcAutoAxis();
     yAxisLab2.CalcAutoAxisLabels();

     yAxis3.CalcAutoAxis();
     yAxisLab3.CalcAutoAxisLabels();
     yAxis3.SetAxisIntercept(pTransform3.GetStopX());
     yAxis3.SetAxisTickDir(ChartObj.AXIS_MAX);

     this.UpdateDraw();
 }
```



It may be that you want to specify date/times for the starting and ending values of the x-axis, instead of a simple index. In that case you use the EventCoordinates.TimeScaleStart and EventCoordinates.TimeScaleStop properties. Those methods use a binary search algorithm to search for the ChartEvent closest to the desired date.time. It then uses the ChartEvent at that index to establish the x-axis scale. The code below is extracted from the **ChartEventExamples.OHLCEventChart** example program.

[C#]

```
public void UpdateScaleAndAxesUsingDates(int startindex)
{
    ChartCalendar startdate = (ChartCalendar) datastartdate.Clone();
    ChartCalendar  stopdate = new ChartCalendar();
    startdate.Add(ChartCalendar.DAY_OF_YEAR,startindex);
    stopdate = (ChartCalendar) startdate.Clone();
    stopdate.Add(ChartObj.MONTH,1);

    pTransform1.TimeScaleStart = startdate;
    pTransform1.TimeScaleStop = stopdate;

    pTransform2.TimeScaleStart = startdate;
    pTransform2.TimeScaleStop = stopdate;

    pTransform3.TimeScaleStart = startdate;
    pTransform3.TimeScaleStop = stopdate;

    xAxis1.CalcAutoAxis();
    yAxis1.CalcAutoAxis();
    xAxisLab1.CalcAutoAxisLabels();
    yAxisLab1.CalcAutoAxisLabels();

    xAxis2.CalcAutoAxis();
    xAxis2.SetAxisIntercept(pTransform2.GetStopY());
    xAxis2.SetAxisTickDir(ChartObj.AXIS_MAX);

    yAxis2.CalcAutoAxis();
    yAxisLab2.CalcAutoAxisLabels();
```

```
    yAxis3.CalcAutoAxis();
    yAxisLab3.CalcAutoAxisLabels();
    yAxis3.SetAxisIntercept(pTransform3.GetStopX());
    yAxis3.SetAxisTickDir(ChartObj.AXIS_MAX);

    this.UpdateDraw();
}
```

If you specify a date/time value which is an exact match for one of the date/values of a ChartEvent, then the ChartEvent Position property becomes the scale value. If it is not an exact match, then the binary search for the closest date/time rounds down to the nearest ChartEvent for the TimeScaleStart property, and rounds up for the TimeScaleStop property.

When using the time/date values for scaling an EventCoordinate system, you cannot set a time/date value not bounded by the range of values found in the attached datasets. In other words, you cannot create datasets using ChartEvents that use time/date values in 2011, and try and scale the x-axis for a range of 2011 to 2013. The largest value you can scale the x-axis for is the time/date value of the ChartEvent with the largest (or latest) time stamp value. The only time/date values which exist in a EventCoordinates based coordinate system are the time stamps of the ChartEvents in the datasets attached to the coordinate systems. Other times and dates do not exist unless you add a ChartEvent containing the date/time to one of the attached datasets.

Polar Coordinate Systems

Class PolarCoordinates

PhysicalCoordinates
|
+-- CartesianCoordinates
|
+-- PolarCoordinates

The magnitude and the polar angle of a point define its position in a chart scaled for polar coordinates. The magnitude can have any value greater than 0.0 and the polar angle any positive or negative value. A polar angle range of 0 to 2 pi radians (0 to 360 degrees) sweeps a complete circle in polar coordinates.

A polar coordinate system uses a Cartesian coordinate system scaled for plus/minus the polar magnitude. The following equations convert from polar coordinates to Cartesian coordinates.

x = magnitude * cos (angle)

y = magnitude * sine (angle)

magnitude	Polar coordinate magnitude
angle	Plot coordinate angle
x	Cartesian x-coordinate
y	Cartesian y-coordinate

The **PolarCoordinates** class is an extension of the **CartesianCoordinates** class and it automatically handles these conversions. The only important parameter that needed for the creation of a **PolarCoordinates** object is the polar magnitude, since the polar angle always has a range 0 to 2 pi radians (0 to 360 degrees).

PolarCoordinates constructors

The first way to create a **PolarCoordinates** object is to use the constructor that specifies the polar magnitude directly.

[C#]

```
double polarmagnitude = 5.0;
PolarCoordinates polarscale = new PolarCoordinates(polarmagnitude);
```

Or you can use an auto-scale routine to analyze a dataset and select the appropriate polar magnitude.

[C#]

```
double []angleData = {.20,.60,1.40,1.70,2.50,4.0,5.0, 6.0};   // In Radians
double []magnitudeData = { 20, 33, 44, 55, 46, 33, 54, 64};
SimpleDataset dataset = new SimpleDataset("Control", angleData, magnitudeData);
PolarCoordinates pPolarTransform = new PolarCoordinates();
pPolarTransform.AutoScale(dataset, ChartObj.AUTOAXES_FAR);
```

Antenna Coordinate Systems

Class AntennaCoordinates

PhysicalCoordinates

|

+-- CartesianCoordinates

|

+-- AntennaCoordinates

An antenna coordinate's point is defined by its radial and angular values. The radial and angle values can be positive or negative. An angle range of 0 to 360 degrees clockwise, starting at 12:00, sweeps a complete circle in antenna coordinates.

AntennaCoordinates are defined by specifying a starting and ending value for the radius. Unlike a polar chart, these values can be positive or negative. Antenna coordinates always have an angular range 0 to 360 degrees.

AntennaCoordinates constructors

The first way to create a **AntennaCoordinates** object is to use the constructor that specifies the minimum and maximum values of the radius: **AntennaCoordinates(minvalue, maxvalue).**

[C#]

```
double minvalue = -40;
double maxvalue = 20;

AntennaCoordinates antennacoords = new AntennaCoordinates (minvalue, maxvalue);
```

Or you can use an auto-scale routine to analyze a dataset and select the appropriate antenna radius limits.

[C#]

```
double []angleData = {0, 30, 60, 90, 120, 150, 180};   // In degrees
double []radiusData = { -35, -31, -5, 12, 14, -14, -30};
SimpleDataset dataset = new SimpleDataset("Control", angleData, radiusData);
AntennaCoordinates antennacoords = new AntennaCoordinates ();
antennacoords.AutoScale(dataset, ChartObj.AUTOAXES_FAR);
```

Miscellaneous Coordinate System Topics

Inverted Coordinate Systems

Charts that use linear, logarithmic and time coordinate systems usually follow the convention that values increase as you move from left to right and from bottom to top. This is not always the case though. Many standard charts that users want to reproduce on the computer have the x-scale, the y-scale, or both, increase as you move from right to left and from top to bottom.

Invert the x- and/or y-scales by swapping the scale starting and ending vaues in the call to the **CartesianCoordinates** or the **TimeCoordinates** constructor.

Example of inverted x-scale using the CartesianCoordinates constructor

[C#]

```
double xMin = -5;
double xMax = 15;
double yMin = 0;
double yMax =  15;
```

```
CartesianCoordinates simpleScale;
simpleScale = new CartesianCoordinates(xMax, yMin, xMin, yMax);
```

Use the **CartesianCoordinates.SetCoordinateBounds** method in the same manner. The example below inverts the y-scale.

```
simpleScale.SetCoordinateBounds(xMin, yMax, xMax, yMin)
```

Invert the x- and y-scale of a **TimeCoordinates** object in an analogous fashion.

Example of inverted scaling using a TimeCoordinates constructor

[C#]

```
ChartCalendar xMin = new ChartCalendar(1996, ChartObj.FEBRUARY,  5);
ChartCalendar xMax = new ChartCalendar(2002, ChartObj.JANUARY,  5);
double yMin = 0;
double yMax =  15;

TimeCoordinates simpleTimeScale;
simpleTimeScale = new TimeCoordinates(xMax, yMin, xMin, yMax);
```

Use the **TimeCoordinates.SetCoordinateBounds** method in the same manner. The example below inverts the y-scale.

[C#]

```
TimeCoordinates simpleTimeScale = new TimeCoordinates();
simpleTimeScale.SetCoordinateBounds(xMin, yMax, xMax, yMin);
```

The auto-scale functions always create scales that increase from left to right, and bottom to top. This does not exclude the use of the auto-scale functions when creating inverted axes. After an auto-scale function creates the initial x- and y-scales, either or both can be inverted by using the **CartesianCoordinates** or **TimeCoordinates InvertScaleX** or **InvertScaleY** methods.

Example of inverting a scale created using the auto-scale methods

[C#]

```
double [] xData = {2,3,4,5,6,7,8,9};
double [] yData = { 22, 33, 44, 55, 46, 33, 25, 14};

SimpleDataset dataset = new SimpleDataset("Sales", xData, yData);
CartesianCoordinates simpleScale  = new CartesianCoordinates();
simpleScale.AutoScale(dataset);
simpleScale.InvertScaleY();
```

5. The Chart View

ChartView

The starting point of a chart is the **ChartView** class. The **ChartView** class derives from the **Windows.UI.Xaml.Controls.UserControl** class. The **ChartView** class contains a collection of all the chart objects displayed in the chart and will automatically update all chart objects when the underlying window moves, resizes, or otherwise needs to redraw in response redraw event. Since a **ChartView** derived window is a **UserControl**, it can also be used as a container for other components.

UserControl

+-- ChartView

The **ChartView** class has only one constructor with no arguments.

ChartView constructor

```
[C#]
public class ChartView : UserControl
```

All chart objects that have a graphical representation, i.e. that consist of lines, bars, arcs, text, etc., are subclasses of the **GraphObj** abstract base class. This includes all of the axis classes, axis label classes, plot classes, text classes and legend classes among others. You must explicitly add objects of this type to the **ChartView** object, using the **ChartView.AddChartObject** method, after they have been created and initialized. Otherwise, the object will not be included in the draw list of the **ChartView**. The example below adds an axis object to the **ChartView** draw list.

ChartView.AddChartObject example (extracted from the example program ScatterPlots, class LabeledDatapoints)

[C#]

```
private void InitializeChart(ChartView chartVu)
{
.
.
.

SimpleDataset Dataset1 = new SimpleDataset("First",x1,y1);
CartesianCoordinates pTransform1 = new CartesianCoordinates( ChartObj.LINEAR_SCALE,
ChartObj.LINEAR_SCALE);
pTransform1.AutoScale(Dataset1, ChartObj.AUTOAXES_FAR, ChartObj.AUTOAXES_FAR);
```

```
pTransform1.SetGraphBorderDiagonal(0.15, .15, .90, 0.7) ;

Background gbackground = new Background(pTransform1, ChartObj.GRAPH_BACKGROUND,
Colors.LightSkyBlue);
chartVu.AddChartObject(gbackground);

LinearAxis xAxis = new LinearAxis(pTransform1, ChartObj.X_AXIS);
chartVu.AddChartObject(xAxis);
.
.
.
```

Rendering Order of GraphObj Objects

The z-order of a graph object defines the order in which it is rendered to the screen. Objects with a low z-order are rendered first, and underneath, objects with a high z-order. This is critically important, because you want plot objects (line plots, bar plots, etc.) on top of background objects. Graph objects which have the same z-order are plotted in the order they are added to the ChartView.

Each **GraphObj** object has a default z-order value, summarized below.

Base Class	Default z-order value	Comments
Background	10	Backgrounds are drawn first. A plot area background has a z-value of 10 and a graph area background has a z-value of 9, forcing graph area backgrounds to be drawn first.
ChartGrid	40	A z-value of 40 places grids under most other graph objects. If you want grids on top change the z-value to 150.
GraphObj	50	The default value for graph objects if not explicitly changed in the subclass.
ChartText	50	The default value for text objects.
ChartPlot	50	The default value for plot objects which includes **SimplePlot**, **GroupPlot**, **ContourPlot**, **PolarPlot,** and **AntennaPlot** objects.
Axis	100	Chart axes are drawn after data plots
AxisLabels	100	Axes labels are drawn with same priority as axes

Legend	150	Legend objects usually sit on top of all other graph objects and are drawn last

You can change the default z-order value on an object-by-object basis. Call the **GraphObj.SetZOrder** method to change the z-order for any given object.

The example below sets the z-order value of the x-axis to 30, changing the drawing order so that the x-axis draws before, and is therefore underneath, any **ChartGrid** and **ChartPlot** objects in the view.

[C#]

```
private void InitializeChart(ChartView chartVu)
{
.
.
.

LinearAxis xAxis = new LinearAxis(pTransform1, ChartObj.X_AXIS);
xAxis.SetZOrder(30);
chartVu.AddChartObject(xAxis);
```

Dynamic or Real-Time Updates of Chart Objects

If you want to change the properties of one or more **GraphObj** derived objects displayed in the current graph, just go ahead and change them using the appropriate Get and Set methods, or properties. Once you change all of the properties that you want, call the **ChartView.UpdateDraw** method. This will force the **ChartView** object to update, redrawing every object in its draw list. See the example below.

[C#]

```
private void InitializeChart(ChartView chartVu)
{
.
.
.

LinearAxis xAxis = new LinearAxis(pTransform1, ChartObj.X_AXIS);
chartVu.AddChartObject(xAxis);
.
.
.
xAxis.SetColor(Colors.Red);
chartVu.UpdateDraw();
```

You can change the values of a dataset, or even change the complete dataset of a chart plot object. Changing the values of the dataset will not show in the current graph until the

ChartView redraws itself. Call the **ChartView.UpdateDraw** method to force a redraw. See the example program DynamicCharts. The auto-scale methods are not automatically invoked if you change the values of dataset. If you want the graph to rescale taking into account the new data values you must call the appropriate autoscale methods of the coordinate system and of the related axes objects.

The chart classes that are NOT subclasses of **GraphObj** do not have a physical representation in a graph, so do not try to add them to the **ChartView** draw list. This includes the coordinate conversion classes, the dataset classes and all of the utility classes. The **GraphObj** and **ChartView** classes use these utility classes for coordinate conversions, data storage, math calculations and I/O.

Delete a specific chart object from the **ChartView** draw list using the **ChartView.DeleteChartObject** method. Clear the entire draw list using the **ChartView.ResetChartObjectList** method. You can leave an object in the **ChartView** draw list but disable its display by calling that objects **GraphObj.SetChartObjEnable** method. If you disable an object, you will still need to call the **ChartView.UpdateDraw** method to redraw the chart without that object.

Placing Multiple Charts in a ChartView

One way to create multiple charts is to create multiple instances of the **ChartView** class and add each **ChartView** object to a UWP layout panel, such as StackPanel, WrapPanel, DockPanel, and Grid. A UWP layout manager manages the position and size of each **ChartView**. Another way is to place multiple charts in the same **ChartView** object. This makes it easier to guarantee alignment between the axes of separate graphs. The trick to doing this is to create separate coordinate system objects (**CartesianCoordinates**, **TimeCoordinates, PolarCoordinates,** or **AntennaCoordinates**) for each chart, and to position the plot area of each coordinate system so that they do not overlap. Use one of the coordinate systems **SetGraphBorder**… methods. Many of the examples use this technique, including GroupBarPlotChart, DoubleBarPlot, OHLFinPlot, FinOptions, DynPieChart, PieAndLineChart and PieAndBarChart.

Multiple charts in a ChartView example (extracted from the example program FinancialExamples, class OHLCChart)

[C#]

```
pTransform1 = new TimeCoordinates();
pTransform1.SetGraphBorderDiagonal(0.1, .15, .90, 0.6) ;

pTransform2 = new TimeCoordinates();
pTransform2.SetGraphBorderDiagonal(0.1, .7, .90, 0.875) ;
```

Multiple Coordinate Systems in the Same Chart

Often a chart needs more than one coordinate system to in order to support multiple x- and y-axes, each with different scales. As in the preceding section, this involves creating multiple coordinate systems. The plot areas for the coordinate systems can occupy separate space in the chart view, or they can overlap at the exact same position. As in the previous section, the position of each coordinate systems plot area in the chart view is set using one of the coordinate systems **SetGraphBorder**… methods. Many of the examples use this technique, including OHLFinPlot, MultiAxes, LinearAxes, LogAxes and DateAxes1 and DateAxes2.

Multiple coordinate systems in a ChartView example

[C#]

```
double xMin1 = -5;
double xMax1 = 15;
double yMin1 = 0;
double yMax1 =  105;
CartesianCoordinates pTransform1 =
           new CartesianCoordinates(xMin1, yMin1, xMax1, yMax1);
pTransform1.SetGraphBorderDiagonal(0.1, .15, .90, 0.6) ;

double xMin2 = -50;
double xMax2 = 150;
double yMin2 = 0;
double yMax2 =  1050;
CartesianCoordinates pTransform2 =
           new CartesianCoordinates(xMin2, yMin2, xMax2, yMax2);
pTransform2.SetGraphBorderDiagonal(0.1, .15, .90, 0.6) ;
```

ChartView Object Resize Modes

Every **GraphObj** object has absolute size properties, such as font size or line thickness. Resize a window and these absolute size parameters are NOT changed. No matter how you resize a chart, if you set a text object to a font size of 10, the text object will always return a font size of 10, *regardless if the text now appears larger or smaller*. Instead, the value of the **ResizeMultiplier** adjusts to represent the proportional change in the window size. In calculating the font size and the line thickness, the current size properties are multiplied by the **ResizeMultiplier**. The initial value of the **ResizeMultiplier** is 1.0 and the objects size properties correspond exactly to the initial settings. Shrink the **ChartView** window and the **ResizeMultiplier** for each object is set to a value that is less than 1.0. Enlarge the window and the **ResizeMultiplier** is set to a value greater than 1.0. The **ChartView** class has three resize modes that it can use to resize graph objects placed in the graph.

NO_RESIZE_OBJECTS — The **ResizeMultiplier** stays fixed at 1.0. Resizing the graph window does not affect the size and thickness of the charts graph objects. Text will stay the same size and lines will stay the same thickness. The overall chart shrinks; it is just that size of the chart text and the thickness of the chart lines do not change. Resize the window small enough and the chart text will overlap and the lines used to draw the chart will look thick when compared to the chart size.

AUTO_RESIZE_OBJECTS — Resizing the graph window causes the size and thickness of the charts graph objects to resize. The auto-resize algorithm looks at which dimension changed the most (x or y) and uses the larger of the two changes to calculate new sizes for lines and text. Text and lines will shrink the same percentage. Resize the chart window with a minimum change in the charts aspect ratio, the change in the chart size will be very close to the change in the font size and line thickness. If the charts aspect ratio changes drastically, the font size and line thickness will resize to reflect the dimension that was *reduced* the most. This minimizes the condition where text overlaps, though it may make the text unreadable if the chart size changes from large to a small with a large aspect ratio change.

MANUAL_RESIZE_OBJECTS — The **ResizeMultiplier** for each object has an initial value of 1.0. Unlike the NO_RESIZE_OBJECTS mode, the value can be changed. The programmer must explicitly set the **ResizeMultiplier** for any objects requiring a size change, using the **GraphObj.SetResizeMultiplier** method.

Set the ResizeMode to automatically text and line objects when the ChartView size changes.

```
chartVu.SetResizeMode(ChartObj.AUTO_RESIZE_OBJECTS);
```

ChartView View Modes

A **ChartView** window can interact with a parent container, creating a couple of interesting view modes. The first is to place the **ChartView** object in a App window, and give it an explicit size, as in the example below.

```
    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}"
Margin="10,10,10,10">
        <FlipView Margin="10,10,10,10">

            <my:ChartView x:Name="fixedsizeframe1" Height="600" Width="800"
HorizontalAlignment="Left" Margin="10,10,10,10" VerticalAlignment="Top"/>
```

```
        </FlipView>
    </Grid>
```

Size the **parent** window smaller than the **ChartView** size and the window clips out the portion of the **ChartView** not viewable. Size the parent window larger than the **ChartView**, and extra space is left around the chart. The size of the **ChartView** remains unchanged in both cases.

If you do not give the **ChartView** an explicit size, the UWP Grid layout panel will resize the **ChartView** to fit into the assigned grid element.

```
    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}"
Margin="10,10,10,10">
        <FlipView Margin="10,10,10,10">
            <my:ChartView x:Name="resizeableobjects1" HorizontalAlignment="Stretch"
Margin="10,10,10,10" VerticalAlignment="Stretch"/>
            <my:ChartView x:Name="fixedsizeobjects1" HorizontalAlignment="Stretch"
Margin="10,10,10,10" VerticalAlignment="Stretch"/>

        </FlipView>
    </Grid>
```

In this case, when the parent window is resized, the **ChartView** will also resize. Whether or not the text and line objects in the chart resize along with the chart depends on the value of the **ResizeMode**, discussed in the previous section.

Another interesting technique is to place a fixed size **ChartView** object in a UWP **ScrollViewer** control. The size of the **ChartView** window can much larger than the window size. Place the chart in a ScrollViewer control and set the scrollbar options to Auto. The auto-scroll feature of the ScrollViewer displays scroll bars when the chart object is larger than the containing form. The scroll bars permit the user to pan left, right, up and down to view the portion of the **ChartView** window that is outside of the clipping limits of the window. See the ResizeExamples.ScrollPanelPolarLine tab of the ResizeExamples program for an example.

```
    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}"
Margin="10,10,10,10">
        <FlipView Margin="10,10,10,10">
            <ScrollViewer  Height="600"  Width="800" VerticalScrollBarVisibility="Auto"
VerticalScrollMode="Auto" HorizontalScrollBarVisibility="Auto"
HorizontalScrollMode="Auto">
                <my:ChartView x:Name="fixedsizescrollable1" HorizontalAlignment="Left"
Height="1000" Margin="10,10,10,10" VerticalAlignment="Top" Width="1248"/>
            </ScrollViewer>
        </FlipView>
    </Grid>
```

There are many more variations. Just remember that the **ChartView** class is a **UserControl** derived class and it can be used anywhere a **UserControl** object can be used, utilizing standard or specialized UWP layout managers.

Finding Chart Objects

The **ChartView** class is the central container class of the chart library. It keeps track of all of the objects in the chart. It includes a routine that can compare a test point against all of the objects in the chart and return an instance of an object that intersects the test point. The search can be restricted to a class and all subclasses of the specified class.

FindObj method

```
[C#]
public GraphObj FindObj(
   Point2D testpoint,
   string classname
);
```

Parameters

If the graph has multiple overlapping objects of the same type, you can return the n^{th} object intersecting the test point.

```
[C#]
public GraphObj FindObj(
   Point2D testpoint,
   string classname,
   int nthhit
);
```

testpoint	The current position of the mouse in UWP device coordinates.
classname	The class name of the base class that is used to filter the desired class objects. The string "ChartPlot" would cause the routine to return only objects derived from the **ChartPlot** class.
nthhit	Specifies to return the n^{th} object that intersects the test point. A value of 0 signifies that the first object found is returned, a value of 1 specifies that the second item found is returned, and so on.

The function returns a reference to the found object, or null if unsuccessful.

6. Colors, Gradients and Backgrounds

Class ChartAttribute

ChartObj
|
+-- ChartAttribute

All graphical object derived from our abstract **GraphObj** class include an instance of the **ChartAttribute** class. This class encapsulates common graphical line and fill style characteristics into a single class. If a graphical object is line based, it can have a line color, line style and a line thickness. Line based graphical objects include line-based plots (SimpleLinePlot, MultiLinePlot, OHLCPlot) all types of axes, and all types of text. If an object is area based, it can have a solid fill color, or a fill gradient, or any other type of Brush derived object you can define. The fill color fills the interior of the area object. Most area fill objects also use line attributes to define the color and line thickness of the outline of the area object. A bar can have an outline color different from the interior fill color. All of the bar graph plot types, scatter plot types, and pie charts are example of graphical objects which use the area fill solid color

ChartAttribute constructors

Use the constructor below for simple line and fill attributes. There are similar constructors with fewer parameters if all you want to do is set a line color, or a line color with a line thickness.

[C#]
```
public ChartAttribute(
   Color rgbcolor,
   double rlinewidth,
   DashStyle nlinestyle,
   Color rgbfillcolor
);
```

rgbcolor	The primary line and text color.
rlinewidth	The line width for all lines
nlinestyle	The line style for all lines.
rgbfillcolor	The fill color for solid objects

[C#]

```
ChartAttribute attrib1 =
           new ChartAttribute (Colors.Blue, 3,ChartObj.ChartObj.LS_SOLID);
SimpleLinePlot thePlot1 =
           new SimpleLinePlot(pTransform1, Dataset1, attrib1);
thePlot1.SetLineStyle(DashDot);
chartVu.AddChartObject(thePlot1);
```

All of the **ChartAttribute** constructors assume you are using a solid area fill color. If you want to use a gradient, you need to attach a **ChartGradient** object to the **ChartAttribute**. The **ChartGradient** object will specify the defining range of colors for the gradient, the breakpoints for the gradient colors, and the mapping mode for mapping the breakpoints to the current chart.

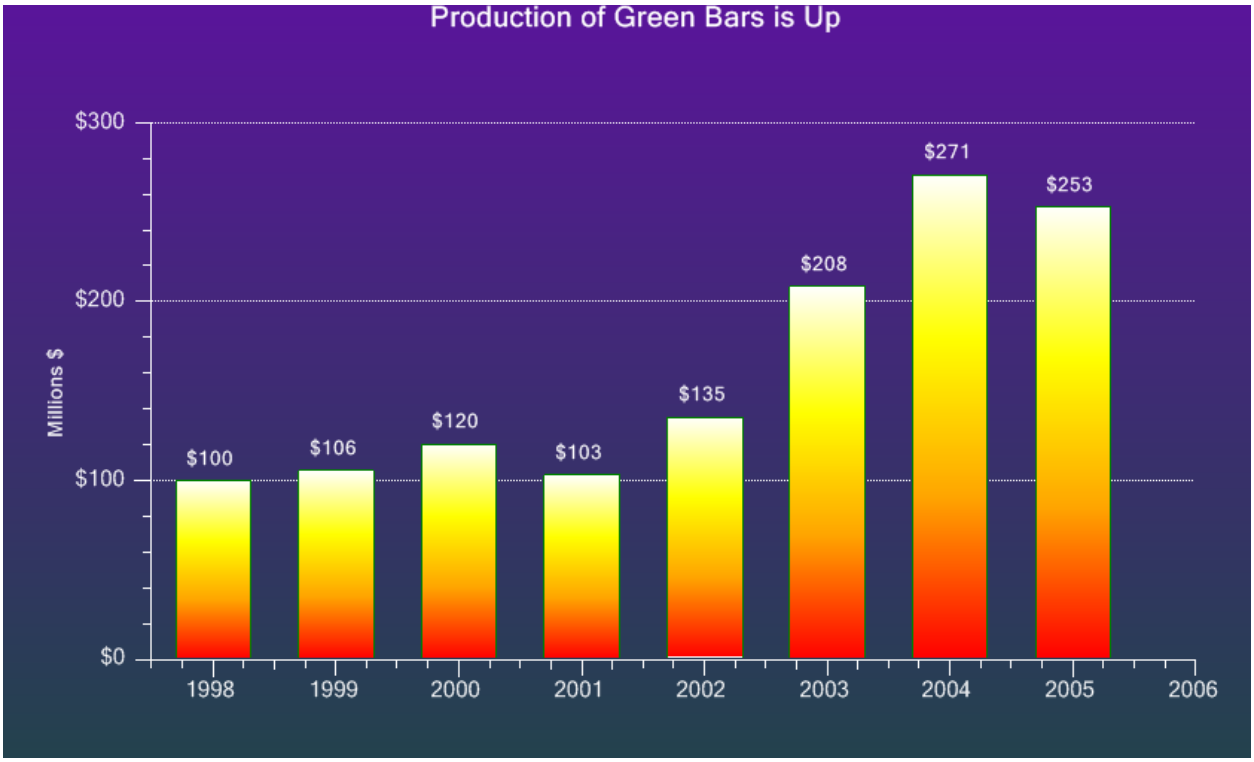
Class ChartGradient

ChartObj
|
+-- ChartGradient

All **ChartAttribute** objects include a reference to a **ChartGradient** object. Normally this reference is null, signifying that line and area fills work exactly the same as before. If the **ChartGradient** reference is not null, the color definitions in the **ChartGradient** take precedence over the fill color of the **ChartAttribute**. Any area fill object can have a gradient of two or more colors mapped to it. The colors are mapped to the area fill object using an array of breakpoints, one for each color, that define the transition points for one color to the next. The values of the breakpoints are interpreted according to one of four different mapping modes:

GRADIENT_MAPTO_OBJECT
In this mapping mode, the breakpoints are expected to be in the range of 0.0 to 1.0. The break points are applied as percentages to the area fill object. The value 0.0 corresponds to the start of the area fill object and the value 1.0 corresponds to the end of the area fill object. It does not matter how large or small the area fill object is, all of the gradient colors will map to that object. This mapping mode would normally used with just two colors, though it will work with an unlimited number of colors.

GRADIENT_MAPTO_OBJECT applied to a simple bar graph

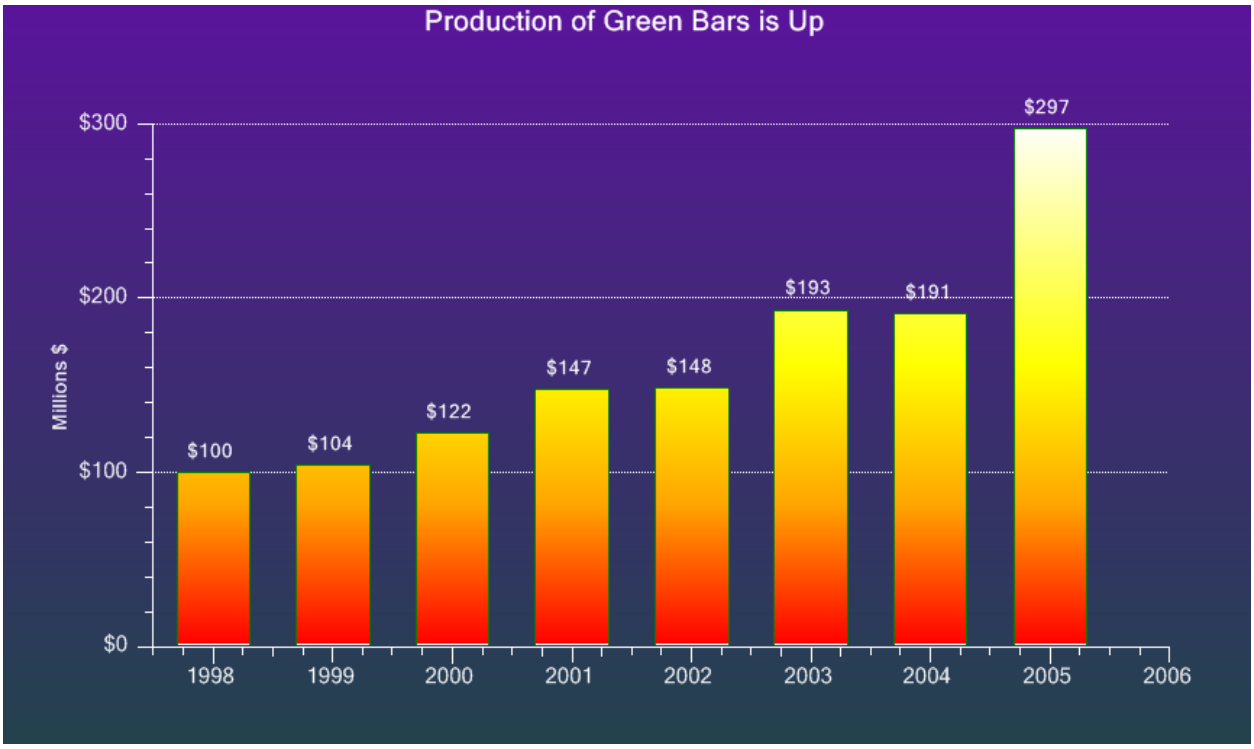


Note how in this example, each bar displays the full range of colors (red, orange, yellow, and white), regardless of the bar height.

GRADIENT_MAPTO_PLOT_PHYSICAL_COORDINATES

In this mapping mode, the breakpoints are expected to be in the range of the physical coordinate system of the plot area of the chart. If the y-scale of the coordinate system has been scaled for 0 – 50,000, then the breakpoints are expected to be in the range of 0-50,000. This allows for the most interesting gradient effects. If you define your gradient breakpoints as extending from 0-50,000, and you plot a bar that is only 10,000 high, only the lower 20% of the gradient will be visible. This way, you can have bars change color as they increase in value. The best analogy would be if you were plotting temperature in a bar graph. As the temperature increases, and the height of the bar, the bar would display as a color gradient. The color gradient would transition from a dull red color, through orange, yellow and finally white, representing the highest color breakpoint.

GRADIENT_MAPTO_PLOT_PHYSICAL_COORDINATES applied to a simple bar graph



Note how in this example, the range of colors in each bar (red, orange, yellow, and white), depends on the bar height.

GRADIENT_MAPTO_PLOT_NORMALIZED_COORDINATES
In this mapping mode, the breakpoints are expected to be in the range of 0.0 to 1.0. The break points are applied as percentages to the plot area. The value 0.0 corresponds to the start of the plot area and the value 1.0 corresponds to the end of the plot area. Unlike the GRADIENT_MAPTO_OBJECT mapping mode, a small area fill object will not show all of the colors of the gradient. Only an area fill object the size of the plot area would show all of the colors. Otherwise, it can be used much the same as the GRADIENT_MAPTO_PLOT_PHYSICAL_COORDINATES, mapping mode, except in this case you are using normalized coordinates (0.0 – 1.0) instead of physical coordinates.

GRADIENT_MAPTO_GRAPH_NORMALIZED_COORDINATES
Much the same as the GRADIENT_MAPTO_PLOT_PHYSICAL_COORDINATES, except that in this case the breakpoints are applied to the entire graph area, not the plot area.

ChartGradient constructors

Use the constructor below for simple line and fill attributes. There are similar constructors with fewer parameters if all you want to do is set a line color, or a line color with a line thickness.

[C#]

```
public ChartGradient(
   PhysicalCoordinates transform,
   int gradmode,
   Color[] gradcolors,
   double[] gradbreak,
   int graddir
);

public ChartGradient(
   int gradmode,
   Color[] gradcolors,
   double[] gradbreak,
   int graddir
);
```

gradcolors An array of colors used to define the gradient.

gradbreak An array of gradient breakpoints (one for each color). The range of values depends on the mapping mode. For the GRADIENT_MAPTO_OBJECT, GRADIENT_MAPTO_PLOT_NORMALIZED_COORDINATES, and GRADIENT_MAPTO_GRAPH_NORMALIZED_COORDINATES modes, the first value of the array should always be 0.0 and the last value should always be 1.0.

graddir The direction of the gradient in degrees. At 0 degrees, the direction is 3:00. Positive degrees rotate clockwise. When used with the *GRADIENT_MAPTO_PLOT_PHYSICAL_COORDINATES* mapping mode, always make degrees even divisible by 90.

gradmode *The mapping mode of the breakpoints to the gradient area.* Use one of the gradient mode constants: GRADIENT_MAPTO_OBJECT, GRADIENT_MAPTO_PLOT_PHYSICAL_COORDINATES, GRADIENT_MAPTO_PLOT_NORMALIZED_COORDINATES, or GRADIENT_MAPTO_GRAPH_NORMALIZED_COORDINATES.

transform The physical coordinate system of the graph object. The coordinate system is for the GRADIENT_MAPTO_PLOT_PHYSICAL_COORDINATES, GRADIENT_MAPTO_PLOT_NORMALIZED_COORDINATES, and GRADIENT_MAPTO_GRAPH_NORMALIZED_COORDINATES mapping modes.

The example below uses the GRADIENT_MAPTO_PLOT_PHYSICAL_COORDINATES mapping mode to map four colors to the physical coordinates of the plot area.

[C#]

```
ChartAttribute attrib1 = new ChartAttribute(Colors.Green, 1, ChartObj.LS_SOLID,
Colors.Yellow);
Color[] barcolors = { Colors.Red, Colors.Orange, Colors.Yellow, Colors.White };


double[] barbreakpoints = { 0.0, 100, 200, 300 };
int gradmode = ChartGradient.GRADIENT_MAPTO_PLOT_PHYSICAL_COORDINATES;
ChartGradient cg = new ChartGradient(pTransform1, gradmode, barcolors, barbreakpoints);
```

```
attrib1.Gradient = cg;

attrib1.SetFillFlag(true);
SimpleBarPlot thePlot1 = new SimpleBarPlot(pTransform1, Dataset1,
ChartCalendar.GetCalendarWidthValue(ChartObj.MONTH, 8), 0.0,
                attrib1, ChartObj.JUSTIFY_CENTER);
```

The example below uses the GRADIENT_MAPTO_OBJECT mapping mode to map four colors to each bar of the bar plot, regardless of the bar size.

[C#]

```
ChartAttribute attrib1 = new ChartAttribute (Colors.Green, 0,ChartObj.LS_SOLID,
Colors.Green);
Color [] barcolors  = {Colors.Red, Colors.Orange, Colors.Yellow, Colors.White};
double [] barbreakpoints = {0.0, 0.33, 0.66, 1.0};

int gradmode = ChartGradient. GRADIENT_MAPTO_OBJECT;
ChartGradient cg = new ChartGradient(pTransform1, gradmode, barcolors, barbreakpoints,
-90 );
attrib1.Gradient = cg;
```

Class Background

Two rectangular areas act as a backdrop for the other graphical elements of a chart. The first is the rectangle formed by the **ChartView** class. This rectangle is the *graph area* and all elements of the chart (axes, labels, plots, titles, etc.) are within its bounds. The second area is the plot area. That area is within the graph area. Its position within the graph area is set using one of the **PhysicalCoordinates.SetGraphBorder…** methods: **SetGraphBorderDiagonal**, **SetGraphBorderFrame** or **SetGraphBorderInsets**. Typically the chart plot objects (line plots, bar plots, scatter plots, etc.) are clipped to the plot area. Other chart objects (axes, axes labels, titles, etc.) need to reside outside of the plot area and these objects clip to the graph area. The plot area can have a background different from that of the graph background. Often a contrast between the graph area background and the plot area background produces a more visually pleasing chart.

GraphObj
|
+-- Background

The **Background** class paints the graph area background or the plot area background. One instance of the class can only paint one area, either the graph area or the plot area. If you want unique fill properties for both, you need to create two instances of the class. The **Background** class uses one of the following techniques to fill the background:

- solid color
- simple color gradient defined using two RGB colors
- user-defined gradient supplied as a UWP **LinearGradientBrush** object
- user-defined drawing brush supplied as a UWP **Brush** object

Background constructors

Use this constructor to fill the background with a single color.

Use this constructor to fill the background with the gradient defined using the *startcolor* and *stopcolor* arguments.

```
[C#]
public Background(
   PhysicalCoordinates transform,
   int bgtype,
   Color startcolor,
   Color stopcolor,
   int dir
);
```

Use this constructor to fill the background with a custom background, a your own LinearGradientBrush, or ImageBrush for example.

```
[C#]
public Background(
   PhysicalCoordinates transform,
   int bgtype,
   Brush gradient
);
```

transform	The coordinate system associated with the chart background. The transform defines where the plot area fits in the graph area.
bgtype	The chart background type. Use one of the chart background type constants: PLOT_BACKGROUND or GRAPH_BACKGROUND. Specifying the PLOT_BACKGROUND type fills the plot area of the chart, while specifying the GRAPH_BACKGROUND type fills the entire graph area of the chart.
gradient	The user defined background brush.
startcolor	Specifies the starting color value of the gradient.
stopcolor	Specifies the ending color value of the gradient.
dir	Specifies the direction of the gradient.

Should you want to use some sort of image as a background for the chart, use the **ChartImage** class and size it to fill the entire view.

The example below defines a simple linear gradient for the graph background area (extracted from the example program SimpleLinePlots, class LineFill)

[C#]

```
pTransform1 = new TimeCoordinates();
pTransform1.AutoScale(DatasetArray, ChartObj.AUTOAXES_FAR, ChartObj.AUTOAXES_FAR);

pTransform1.SetGraphBorderDiagonal(0.15, .1, .92, 0.75);
Background background = new Background(pTransform1, ChartObj.GRAPH_BACKGROUND,
                Colors.Coral, Colors.Blue, ChartObj.Y_AXIS);

chartVu.AddChartObject(background);
```

The example below defines a custom linear gradient for the graph background area.

[C#]

```
  LinearGradientBrush lgb = new LinearGradientBrush();
  GradientStop gs1 = new GradientStop();
  gs1.Color = Colors.LightBlue;
  gs1.Offset = 0;
  GradientStop gs2 = new GradientStop();
  gs2.Color = Colors.Salmon;
  gs2.Offset = 1;

  lgb.GradientStops.Add(gs1);
  lgb.GradientStops.Add(gs2);
  lgb.StartPoint = new Point(0,0);
  lgb.EndPoint = new Point(0, 1);
  Background background = new Background(pTransform1, ChartObj.GRAPH_BACKGROUND, lgb);
  chartVu.AddChartObject(background);
```

7. Axes

Axis

- **LinearAxis**
  - **ElapsedTimeAxis**
  - **PolarAxes**
  - **AntennaAxes**
  - **EventAxis**
- **LogAxis**
- **TimeAxis**

Chart axes describe for the viewer the physical coordinate system used to scale the plot area of a chart. A well-defined, visually appealing chart will display one or more axes with the following characteristics:

- Minimum and maximum values for axes endpoints that are appropriate for the displayed data
- Appropriately spaced axis tick marks that permit the user to easily interpolate by simple inspection data values located between labeled tick marks
- Axis tick mark labels that fall on logical, even intervals
- Flexible axis placement, inside or outside the plot area
- Axes for linear,date/time, elapsed time, logarithmic, polar and antenna, physical coordinate systems

The programmer can explicitly set these characteristics, or they can be calculated automatically based on an analysis of the associated chart data.

The axes of a chart do not define the physical coordinate system of the chart. Rather, the axes provide a visual key to the physical coordinate system. Define the physical coordinate system first using one of the classes derived from **PhysicalCoordinates**. Next, create the axes that reside in the physical coordinate. It is possible to define a physical coordinate system scaled using a xy range of (0-100, 0-100) and create an axis, residing in that coordinate system, that has minimum and maximum values of (0-25). The axis in that case takes up 25% of the chart plot area of the chart. Define the same axis with minimum and maximum values of (0-100) and the axes will span 100% of chart plot area.

A chart axis consists of at least two and usually three parts: the axis line, the axis tick marks, and the axis labels. The axis line extends from the minimum value to the maximum value of the axis. Major tick marks perpendicular to the axis line divide the axis line into sub ranges suitable for labeling. Minor tick marks, also perpendicular to the axis line, further subdivide the space between the major tick marks into even smaller intervals. Axis labels are optional. On one side of a chart there may be an axis with labels and on the other side an axis without labels.

Chart Axes

There are five concrete axis types supported by the **QCChart2D for Windows Apps** library:

Axis Type	Class
Linear	**LinearAxis**
Logarithmic	**LogAxis**
Date/time	**TimeAxis**
ElapsedTime	**ElapsedTimeAxis**
Event	**EventAxis**
Polar	**PolarAxes**
Antenna	**AntennaAxes**

The seven axis types derive directly or indirectly from the **Axis** abstract base class that provides a core set of properties and methods.

All axis objects use the same set of methods, found in the base **GraphObj** class, to set the drawing properties of the lines used to draw the axis line and tick marks. The default values use a black solid line of thickness 1.0. Change the default values using the **GraphObj** methods below.

SetColor method

```
[C#]
public virtual void SetColor(
   Color rgbcolor
);
```

SetLineWidth method

```
[C#]
public virtual void SetLineWidth(
   double linewidth
);
```

SetLineStyle method

```
[C#]
public virtual void SetLineStyle(
   DashStyle linestyle
);
```

rgbcolor	Sets the primary line color for the chart object.
linewidth	Sets the line width, in device coordinates, for the chart object.
linestyle	Sets the line style for the chart object. Use one of the ChartObj line style enumerated constants: LS_SOLID, LS_DASH_4_4, LS_DASH_4_2, LS_DASH_2_2, LS_DOT_1_1, LS_DOT_1_2, LS_DOT_1_4, LS_DOT_1_8, LS_DASH_DOT.

Linear Axes

Class LinearAxis

GraphObj
|
+--Axis
|
+-- LinearAxis

Linear Axis Minimum and Maximum

The axes minimum and maximum are the physical coordinate values that define the starting and ending points of the axis line. It is a mistake to try to invert the axis, i.e. an axis where the scale decreases from left to right, or bottom to top, by setting axis minimum to a value greater than the axis maximum. The software swaps the values if this happens. Create an inverted axis by first defining an inverted physical coordinate system using one of the **PhysicalCoordinates** derived classes. Place the axis in the inverted coordinate system.

The minimum and maximum of a linear axis can assume any numeric values. This differentiates the linear axis from logarithmic, time, polar, and antenna, axes which have specific, valid numeric ranges.

Linear Minor and Major Tick Mark Intervals

Major tick marks perpendicular to the axis line divide the line into sub ranges suitable for labeling. Minor tick marks, also perpendicular to the axis line, further subdivide the space between the major tick marks into even smaller intervals.

The major tick mark interval for a linear axis is set equal to a specified integer number of minor tick intervals, forcing major tick marks to always fall on a minor tick mark.

It is important that the tick mark intervals fall on rounded values appropriate to the physical scale of the chart. It is not appropriate to look at the range (maximum value – minimum value) and divide by some integer. For example, an axis with endpoints –5 to 30 should have a major tick mark interval of 5 or 10, and a minor tick mark interval 1.0. Dividing the axis range (30 – (-5) = 35.0) by 10 will result in a tick interval of 3.5, which is inappropriate for either major or minor

tick intervals. The programmer can calculate the proper tick mark intervals using custom algorithms, or use the automatic methods that are used by default in the axis constructors.

Linear Axis Intercept

An axis resides in a 2-dimensional physical coordinate system. The minimum and maximum values for the axis provide coordinate information for only one dimension; x-coordinates in the case of an x-axis, and y-coordinates in the case of the y-axis. The missing coordinate needed to position the axis is the axis intercept. The axis intercept specifies the y-coordinate position for the x-axis, and the x-coordinate position for the y-axis.

Linear Axis Tick Mark Origin

The axis major and minor tick mark intervals specify the space between adjacent tick marks. A minor tick mark interval of 1.0, and a major tick mark interval of 5.0, may result in major tick marks at 0.0, 5.0, 10.0, 15.0, 20.0, etc. It can also result in major tick marks at –0.88769, 4.11231, 9.11231, 14.11231, 19.11231, etc. Obviously, the first example is the desired tick mark placement. The difference between the two examples is the tick mark starting point, or origin. In the first example, the tick mark origin is 0.0 and in the second case the tick mark origin is –0.88769. The tick mark origin is an important property because often it should not be the minimum value of the axis, but rather some intermediate value between the minimum and maximum value of the axis. In the example above, the data may range from –0.88769 to 19.9 and the chart is to have exactly that range. It is still appropriate that the tick mark origin be set to 0.0, rather than the axis minimum value of –0.88769.

The tick mark origin should reside in the bounds defined by the axis minimum and maximum, inclusive of the endpoints. It does not need to be near an endpoint however. For example, an axis with endpoints –16 to +19 should use a minor tick mark interval of 1.0 or 2.0, a major tick mark interval of 5.0 or 10.0, and a tick mark origin of 0.0. Usually, if the axis minimum and maximum bracket 0.0, i.e. the axis minimum is less than or equal to 0.0 and the axis maximum is greater than or equal to 0.0, the best tick mark origin to use is 0.0.

Creating a Linear Axis

There are two main constructors for **LinearAxis** objects. The first **LinearAxis** constructor assumes that the axis extents match the extents of the underlying coordinate system, *transform*. The second **LinearAxis** constructor sets the axis extents to the specified minimum and maximum values, regardless of the underlying coordinate system.

LinearAxis constructors

```
[C#]
public LinearAxis(
   PhysicalCoordinates transform,
   int axtype
);

public LinearAxis(
   PhysicalCoordinates transform,
   int axtype,
   double minval,
```

```
   double maxval
);
```

transform Places the axes in the coordinate system defined by transform.

axtype Specifies if the axis is an x-axis (X_AXIS), or a y-axis (Y_AXIS).

minval Sets the minimum value for the axis.

maxval Sets the maximum value for the axis.

Other axis properties: minor tick mark spacing, number of minor tick marks per major tick mark, axis intercept, tick mark lengths, tick mark direction and axis tick mark origin are automatically calculated using an auto-axis method. Set these properties explicitly if you need to override the automatically calculated values.

SetAxisIntercept method

```
[C#]
public void SetAxisIntercept(
   double intercept
);
```

SetAxisTicks method

```
[C#]
public void SetAxisTicks(
   double tickorigin,
   double tickspace,
   int nticksperma jor
);

public void SetAxisTicks(
   double tickorigin,
   double tickspace,
   int nminorticksperma jor,
   double minorticklength,
   double majorticklength,
   int tickdir
);
```

intercept Sets the intercept of this axis with the perpendicular axis in physical coordinates.

tickorigin The tick marks start at this value.

tickspace Specifies the spacing between minor tick marks.

nticksperma jor Specifies the number of minor tick marks per major tick mark.

minorticklength	The length of minor tick marks, in UWP device coordinates.
majorticklength	The length of major tick marks, in UWP device coordinates.
tickdir	The direction of the tick marks. Use one of the tick mark direction constants: AXIS_MIN, AXIS_CENTER, or AXIS_MAX.

Use the **SetLineWidth**, **SetLineStyle** and **SetColor** methods to customize the drawing properties of the lines used to draw the axis line and tick marks.

Simple linear axis example

[C#]

```
// Define the coordinate system
double xMin = -5;
double xMax = 15;
double yMin = 0;
double yMax =  105;
CartesianCoordinates simpleScale =
               new CartesianCoordinates(xMin, yMin, xMax, yMax);

// Create the x- and y-axes
LinearAxis xAxis = new LinearAxis(simpleScale, ChartObj.X_AXIS);
LinearAxis yAxis = new LinearAxis(simpleScale, ChartObj.Y_AXIS);

// Create the ChartView object to place graph objects in.
ChartView chartVu  = new ChartView();

// Add the x- and y-axes to the chartVu object
chartVu.AddChartObject(xAxis);
chartVu.AddChartObject(yAxis);
```

Customize the axis by adding the following lines after the creation of the *xAxis* object:

Custom linear axis example

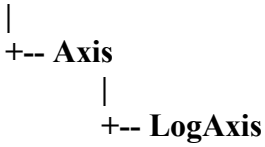
[C#]

```
double xAxisIntercept = -5;
double xAxisOrigin = 0.0;
double xAxisMinorTickSpace = 1.0;
int xAxisMinorTicksPerMajor = 5;
double xAxisMinorTickLength = 5;
double xAxisMajorTickLength = 10;
int xAxisTickDirection = ChartObj.AXIS_MIN;

xAxis.SetAxisIntercept(xAxisIntercept);
xAxis.SetAxisTicks(xAxisOrigin, xAxisMinorTickSpace,
                   xAxisMinorTicksPerMajor, xAxisMinorTickLength,
                   xAxisMajorTickLength, xAxisTickDirection);
```

Logarithmic Axes

Scientific, engineering and financial applications often require the use of logarithmic axe. Logarithmic axes are useful for the display of data that either has a wide dynamic range and/or data that is exponential in nature. Two common examples that use logarithmic scales are hi-fi speaker charts (db vs. log frequency) and stock market charts.

Class LogAxis

GraphObj


The **LogAxis** class is a concrete subclass of the **Axis** class. Use the **LogAxis** class to create a logarithmic axis with logarithmic spacing between the major tick marks (1, 10, 100…), and linear spacing (2, 3, 4, 5…) between the minor tick marks.

Logarithmic Axis Minimum and Maximum

The minimum and maximum values for a logarithmic axis can have any positive value, as long as the maximum is greater than the minimum. Create an inverted axis by first defining an inverted physical coordinate system using one of the **PhysicalCoordinates** derived classes. The axis minimum and maximum do not have to fall on decade intervals, i.e. 0.1 to 10,000 and can assume any positive range, i.e. 0.23 to 13,100 is valid.

Logarithmic Minor and Major Tick Mark Intervals

The major tick marks for a logarithmic axis use an exponential interval. The exponential interval in physical coordinates transforms to a linear interval in the working coordinate system. Below are examples of the major tick mark locations for a logarithmic axis.

Axis Minimum and Maximum	Axis Major Tick Mark Locations
0.1 to 100.0	0.1, 1.0, 10.0, 100.0
20 to 50,000	20, 200, 2000, 20000
10^{-4} to 1.0	10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1.0

The minor tick marks for a logarithmic axis use a linear interval between the tick marks. For example, a major tick mark interval has endpoints of 10 to 100, a logarithmic interval. The minor ticks in-between the 10 and the 100 use a linear interval of 10 and fall at 20, 30, 40, 50, 60, 70, 80, and 90. For the next major tick mark interval, 100 to 1000, the minor tick mark interval becomes 100 and minor tick marks fall at 200, 300, 400, 500, 600, 700, 800, and 900. The minor

tick mark intervals are set equal to the value of the preceding major tick mark interval. If the major tick mark interval uses a non-decade range, for example 3, 30, 300, 30000, the minor tick marks will track the major tick marks. The major tick mark interval of 3 to 30 will use a minor tick mark range of 3, with minor tick marks at 6, 9, 12, 15, 18, 21, 24, and 27.

Logarithmic Axis Intercept

A logarithmic axis has an intercept value, the same as a linear axis. Since the intercept value is specified using the scale of the perpendicular axis, if the perpendicular axis is linear, as in the case of semi-log graphs, the intercept value can be positive, negative, or 0.0. If the perpendicular axis is logarithmic, the intercept value is restricted to a positive range.

Logarithmic Axis Tick Mark Origin

The starting value for the major tick marks does not need to fall at the end of the axis range. For example, the axis may have a range of 0.175 to 195. It would not make sense to start the major tick mark placement at 0.175. The major tick marks would end up placed at 0.175, 1.75, 17.5 and 175. The minor tick marks would make even less sense. A better major tick mark placement is 0.2, 2, 20, and 200. The minor tick marks will also fall on even values.

The logarithmic axis tick mark origin controls the placement of the first major tick mark. The other major and minor tick mark positions are automatically calculated based on the initial position of the first major tick mark.

The tick mark origin must reside in the bounds defined by the axis minimum and maximum, inclusive of the endpoints. It does not need to be near an endpoint however.

LogAxis Constructors

There are two constructors for **LogAxis** objects.

```
[C#]
public LogAxis(
   PhysicalCoordinates transform,
   int axtype
);

public LogAxis(
   PhysicalCoordinates transform,
   int axtype,
   double minval,
   double maxval
);
```

transform Places the axes in the coordinate system defined by transform.

axtype Specifies if the axis is an x-axis (X_AXIS), or a y-axis (Y_AXIS).

minval Sets the minimum value for the axis.

maxval Sets the maximum value for the axis.

The first **LogAxis** constructor assumes that the axis extents match the extents of the underlying coordinate system, *transform*. The second **LogAxis** constructor sets the axis extents to the specified minimum and maximum values, regardless of the underlying coordinate system.

Other axis properties: axis intercept, tick mark lengths, tick mark direction and axis tick mark origin are automatically calculated using an auto-axis method. These properties can be explicitly set if you need to override the automatically calculated values.

SetAxisIntercept method

```
[C#]
public void SetAxisIntercept(
   double intercept
);
```

SetAxisTicks method

```
[C#]
public void SetAxisTicks(
   double tickorigin,
   int nlogtickformat
);

public void SetAxisTicks(
   double origin,
   int nlogtickformat,
   double minorticklength,
   double majorticklength,
   int tickdir
);
```

intercept Sets the intercept of this axis with the perpendicular axis in physical coordinates.

nlogtickformat This parameter specifies which minor tick marks are flagged for labels. Logarithmic axis minor tick mark labels can become very crowded. It is possible to choose values that may overlap or not display. Valid *nlogtickformat* values are:

0	No minor tick mark labels
1	Place a label at tick mark 0 in each decade.
2	Place a label at minor tick marks 1, 3 and 5 in each decade.
3	Place a label at minor tick marks 0, 1, 2, 3 and 5 in each decade.
4	Place a label at minor tick marks 0, 1, 2, 3, 4 and 5 in each decade.
5	Place a label at minor tick marks 0, 1, 2, 3, 4, 5 and 6 in each decade.

6 Place a label at minor tick marks 0, 1, 2, 3, 4, 5, 6 and 7 in each decade.

7 Place a label at minor tick marks 0, 1, 2, 3, 4, 5, 6, 7 and 8 in each decade.

8 Place a label at minor tick marks 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9 in each decade.

minorticklength The length of minor tick marks, in UWP device coordinates.

majorticklength The length of major tick marks, in UWP device coordinates.

ticdir The direction of the tick marks. Use one of the tick mark direction constants: AXIS_MIN, AXIS_CENTER, or AXIS_MAX.

The **SetLineWidth**, **SetLineStyle** and **SetColor** methods are used to customize the drawing properties of the lines used to draw the axis line and tick marks.

Simple log axis example

[C#]

```
double xMin = 0;
double xMax = 1000;
double yMin = 0.2;
double yMax =  2000;
CartesianCoordinates logYScale =
       new CartesianCoordinates(ChartObj.LINEAR_SCALE,  ChartObj.LOG_SCALE);
logYScale.SetCoordinateBounds(xMin, yMin, xMax, yMax);

// Create a linear x-axis and a logarithmic y-axis
LinearAxis xAxis = new LinearAxis (logYScale, ChartObj.X_AXIS);
LogAxis yAxis = new LogAxis (logYScale, ChartObj.Y_AXIS);

// Create the ChartView object to place graph objects in.
ChartView chartVu  = new ChartView();

// Add the x- and y-axes to the chartVu object
chartVu.AddChartObject(xAxis);
chartVu.AddChartObject(yAxis);
```

Should want to customize the axis you can add the following lines after the *yAxis* object is created:

Custom logarithmic axis example

[C#]

```
// Place the y-axis on the right side of the graph with tick marks
// point towards the right.
double yAxisIntercept = 1000;
// Major tick marks at 0.2, 2, 20, 200 and 2000
```

```
double yAxisOrigin = 0.2;
// In addition to major tick marks, labels flagged for some minor tick marks
int yAxisLogFormat = 1;
double yAxisMinorTickLength = 5;
double yAxisMajorTickLength = 10;
int yAxisTickDirection = ChartObj.AXIS_MAX;

yAxis.SetAxisIntercept(yAxisIntercept);
yAxis.SetAxisTicks(yAxisOrigin, yAxisLogFormat, yAxisMinorTickLength,
                   yAxisMajorTickLength, yAxisTickDirection);
```

Date/Time Axes

The date/time axis is used in combination with a **TimeCoordinates** physical coordinate system. The axis major and minor tick marks correspond to the common date/time divisions of second, minute, hour, day, week, month and year. The date/time axes supported with this software are very complex, because they take into account the varying number of days in months and years. The axes also take into account non-continuous date/time scales where a 5-day week is used, or where a full day consists of a specific time interval that can be something less than a 24-hour day.

Note – The **TimeAxis** class is **not** used to create an axis to display elapsed time. Use a **ElapsedTimeCoordinates** system in combination with a **ElapsedTimeAxis** to create an elapsed time axis. It is the **ElapsedTimeAxisLabels** that give the elapsed time axis its time axis look, i.e. 10:30:22.

Class TimeAxis

GraphObj
|
+--Axis
|
+-- TimeAxis

The **TimeAxis** class creates an axis with date/time specific spacing between minor and major tick marks, not necessarily uniform as with a **LinearAxis**. The **TimeAxis** extends the **Axis** class.

Date/Time Axis Minimum and Maximum

The minimum and maximum values for a date/time axis can have any valid date/time value, specified using the class **ChartCalendar**. The axis maximum value should be later in time than the minimum. Create an inverted axis by first defining an inverted physical coordinate system using the **TimeCoordinates** class. The axis minimum and maximum do not have to fall on even time or date intervals and can assume any date compatible with the **ChartCalendar** class. For example:

Starting Date and Time	Ending Date and Time	Range
1/1/1972 00:00:00	1/1/1999 00:00:00	27 years
11/04/1997 8:30:00	11/04/1997 16:00:00	7 hours 30 minutes
11/28/2000 8:31:22	1/14/2001 15:14:33	48 days 6 hours 43 minutes 11 sec

Date/Time Minor and Major Tick Mark Intervals

The predefined date/time axis tick mark constants listed below specify both major and minor tick mark spacing.

Date/Time Axis Tick Mark Constants	Description
TIMEAXIS_50YEAR10YEAR	50 year major tick mark spacing, 10 year minor tick mark spacing
TIMEAXIS_20YEAR5YEAR	20 year major tick mark spacing, 5 year minor tick mark spacing
TIMEAXIS_10YEARYEAR	10 year major tick mark spacing, 1 year minor tick mark spacing
TIMEAXIS_5YEARYEAR	5 year major tick mark spacing, 1 year minor tick mark spacing
TIMEAXIS_YEAR	1 year major tick mark spacing
TIMEAXIS_YEARQUARTER	1 year major tick mark spacing, 1 quarter minor tick mark spacing
TIMEAXIS_YEARMONTH	1 year major tick mark spacing, 1 month minor tick mark spacing
TIMEAXIS_QUARTER	1 quarter major tick mark spacing
TIMEAXIS_QUARTERMONTH	1 quarter major tick mark spacing, 1 month minor tick mark spacing
TIMEAXIS_MONTH	1 month major tick mark spacing

TIMEAXIS_MONTHWEEK	1 month major tick mark spacing, 1 week minor tick mark spacing
TIMEAXIS_MONTHDAY	1 month major tick mark spacing, 1 day minor tick mark spacing
TIMEAXIS_WEEK	1 week major tick mark spacing
TIMEAXIS_WEEKDAY	1 week major tick mark spacing, 1 day minor tick mark spacing
TIMEAXIS_DAY	1 day major tick mark spacing
TIMEAXIS_DAY12HOUR	1 day major tick mark spacing, 12 hour minor tick mark spacing
TIMEAXIS_DAY8HOUR	1 day major tick mark spacing, 8 hour minor tick mark spacing
TIMEAXIS_DAY4HOUR	1 day major tick mark spacing, 4 hour minor tick mark spacing
TIMEAXIS_DAY2HOUR	1 day major tick mark spacing, 2 hour minor tick mark spacing
TIMEAXIS_DAYHOUR	1 day major tick mark spacing, 1 hour minor tick mark spacing
TIMEAXIS_12HOURHOUR	12 hour major tick mark spacing, 1 hour minor tick mark spacing
TIMEAXIS_8HOURHOUR	8 hour major tick mark spacing, 1 hour minor tick mark spacing
TIMEAXIS_4HOURHOUR	4 hour major tick mark spacing, 1 hour minor tick mark spacing
TIMEAXIS_2HOURHOUR	2 hour major tick mark spacing, 1 hour minor tick mark spacing
TIMEAXIS_HOUR	1 hour major tick mark spacing
TIMEAXIS_HOUR30MINUTE	1 hour major tick mark spacing, 30 minute minor tick mark spacing
TIMEAXIS_HOUR15MINUTE	1 hour major tick mark spacing, 15 minute minor tick mark spacing

TIMEAXIS_HOUR10MINUTE	1 hour major tick mark spacing, 10 minute minor tick mark spacing
TIMEAXIS_HOUR5MINUTE	1 hour major tick mark spacing, 5 minute minor tick mark spacing
TIMEAXIS_HOUR2MINUTE	1 hour major tick mark spacing, 2 minute minor tick mark spacing
TIMEAXIS_HOURMINUTE	1 hour major tick mark spacing, 1 minute minor tick mark spacing
TIMEAXIS_30MINUTEMINUTE	30 minute major tick mark spacing, 1 minute minor tick mark spacing
TIMEAXIS_15MINUTEMINUTE	15 minute major tick mark spacing, 1 minute minor tick mark spacing
TIMEAXIS_10MINUTEMINUTE	10 minute major tick mark spacing, 1 minute minor tick mark spacing
TIMEAXIS_5MINUTEMINUTE	5 minute major tick mark spacing, 1 minute minor tick mark spacing
TIMEAXIS_2MINUTEMINUTE	2 minute major tick mark spacing, 1 minute minor tick mark spacing
TIMEAXIS_MINUTE	1 minute major tick mark spacing
TIMEAXIS_MINUTE30SECOND	1 minute major tick mark spacing, 30 second minor tick mark spacing
TIMEAXIS_MINUTE15SECOND	1 minute major tick mark spacing, 15 second minor tick mark spacing
TIMEAXIS_MINUTE10SECOND	1 minute major tick mark spacing, 10 second minor tick mark spacing
TIMEAXIS_MINUTE5SECOND	1 minute major tick mark spacing, 5 second minor tick mark spacing
TIMEAXIS_MINUTE2SECOND	1 minute major tick mark spacing, 2 second minor tick mark spacing
TIMEAXIS_MINUTESECOND	1 minute major tick mark spacing, 1 second minor tick mark spacing
TIMEAXIS_30SECONDSECOND	30 second major tick mark spacing, 1 second minor tick mark spacing

TIMEAXIS_15SECONDSECOND	15 second major tick mark spacing, 1 second minor tick mark spacing
TIMEAXIS_10SECONDSECOND	10 second major tick mark spacing, 1 second minor tick mark spacing
TIMEAXIS_5SECONDSECOND	5 second major tick mark spacing, 1 second minor tick mark spacing
TIMEAXIS_2SECONDSECOND	2 second major tick mark spacing, 1 second minor tick mark spacing
TIMEAXIS_SECOND	1 second major tick mark spacing

Sunday is the first day of the week for 7-day weeks, while Monday is the first day of the week for 5-day weeks.

It may not be immediately obvious, but the major tick marks for date/time scales do not necessarily fall on equal intervals. If the tick marks are set to the TIMEAXIS_MONTHDAY value, there may be 28, 29, 30 or 31 days between each months major tick mark. In most cases, the major tick mark coincides with a minor tick mark. For example, if the TIMEAXIS_MONTHDAY setting is used, the major tick mark for a month falls at the first day of the month, coinciding with the minor tick mark for that day. If the TIMEAXIS_WEEKDAY setting is used, the major tick mark for a WEEK falls at the first day of the week, coinciding with the minor tick mark for that day. Situations where the major and minor tick marks do not coincide involve weeks as minor tick marks. If the TIMEAXIS_MONTHWEEK setting is used, the first day of the month may or may not correspond to the first day of the week (Sunday or Monday). Major tick marks fall on the first day of each month, and minor tick marks fall on the first day of each week.

Combine these irregularities with a 5- or 7-day workweek option, and the non-24 hour day option and you can see that a generalized algorithm to define the positions of tick marks is a difficult task. For example: you are a stock trader and you want to track the price/volume characteristics of a stock from the last 5 minutes of the regular trading day, to the first 5 minutes of the next regular trading day, specifically from 3:55 PM Friday, Sept 29, 2000 to 9:35 AM Monday, Oct 2, 2000. The time range under consideration is only 10 minutes, since the market closes Friday at 4:00 PM and opens Monday at 9:30 PM. The resulting axis should reflect this 10-minute range, even though the actual time elapsed is 3,930 minutes. You should be able to specify the axis minimum and maximum values using the actual dates and times. You also need to set a 5-day workweek mode and establish a day that has a time range of 9:30 AM to 4:00 PM.

Date/Time Axis Intercept

A date/time axis has an intercept value, the same as a linear axis. Since the intercept value is specified using the scale of the perpendicular axis, if the perpendicular axis is linear, the

intercept value can be positive, negative, or 0.0. If the perpendicular axis is logarithmic, the intercept value is restricted to a positive range.

There are three main constructors for **TimeAxis** objects. The first two **TimeAxis** constructors assume that the axis extents match the extents of the underlying coordinate system, *transform*. The third **TimeAxis** constructor sets the axis extents to the specified minimum and maximum values, regardless of the underlying coordinate system.

TimeAxis constructors

```
[C#]
public TimeAxis(
   TimeCoordinates transform
);

public TimeAxis(
   TimeCoordinates transform,
   int ntickmarkbase
);
public TimeAxis(
   TimeCoordinates transform,
   ChartCalendar dstart,
   ChartCalendar dstop
);

public TimeAxis(
   TimeCoordinates transform,
   int axtype

);

public TimeAxis(
   TimeCoordinates transform,
   int axtype,
   int ntickmarkbase
);
public TimeAxis(
   TimeCoordinates transform,
   int axtype,
   ChartCalendar dstart,
   ChartCalendar dstop
);
```

If the constuctor does not explicitly specify whether the axis is for the x- or y-axis, then the software checks the underlying TimeCoordinates system (*transform*), and creates an axis for the dimension of the coordinate system that is time based.

transform	The time coordinate system the axis is placed in. If the starting and ending dates of the axis are not explicitly set, the axis uses the starting and ending dates of the *transform* time-scale.
axtype	The axis types. Use one of the axis type constants: X_AXIS or Y_AXIS..
dstart	The starting date value for the axis.
dstop	The ending date value for the axis

ntickmarkbase This field defines the major and minor tick mark spacing for a time axis. Use one of the Date/time axis tick mark mode constants: TIMEAXIS_YEARMONTH, TIMEAXIS_DAYHOUR for example.

Other axis properties: axis intercept, tick mark lengths, tick mark direction are automatically calculated using an auto-axis method. These properties can be explicitly set if you need to override the automatically calculated values.

SetAxisIntercept method

```
[C#]
public void SetAxisIntercept(
   double intercept
);
```

SetAxisTicksAttributes method

```
[C#]
public void SetAxisTicksAttributes(
   double minorticklength,
   double majorticklength,
   int tickdir
);
```

intercept Sets the intercept of this axis with the perpendicular axis in physical coordinates.

minorticklength Specifies the length of a minor tick mark in UWP device coordinates.

majorticklength Specifies the length of a major tick mark in UWP device coordinates.

tickdir Specifies the direction of the tick marks with respect to axis line. Use one of the following tick direction constants: AXIS_MIN, AXIS_CENTER, AXIS_MAX.

Customize the line and tick mark drawing properties of the axis using the **SetLineWidth**, **SetLineStyle** and **SetColor** methods.

Simple time axis example

[C#]

```
// Define a Time coordinate system
ChartCalendar xMin = new ChartCalendar(1996, ChartObj.FEBRUARY,  5);
ChartCalendar xMax = new ChartCalendar(2002, ChartObj.JANUARY,  5);
double yMin = 0;
double yMax =  105;

TimeCoordinates simpleTimeScale;
simpleTimeScale = new TimeCoordinates(xMin, yMin, xMax, yMax);
```

```
// Create the time axis (x-axis is assumed)
TimeAxis xAxis = new TimeAxis(simpleTimeScale);
// Create the linear y-axis
LinearAxis yAxis = new LinearAxis(simpleTimeScale, ChartObj.Y_AXIS);
```

Custom time axis example

[C#]

```
// Define a Time coordinate system
ChartCalendar xMin = new ChartCalendar(1996, ChartObj.FEBRUARY,  5);
ChartCalendar xMax = new ChartCalendar(2002, ChartObj.JANUARY,  5);
double yMin = 0;
double yMax =  105;
int  xAxisTickMarkFormat = ChartObj.TIMEAXIS_MONTHWEEK;
double xAxisMinorTickLength = 5;
double xAxisMajorTickLength = 10;
int xAxisTickDirection = ChartObj.AXIS_MIN;
TimeCoordinates simpleTimeScale;
simpleTimeScale = new TimeCoordinates(xMin, yMin, xMax, yMax);

// Create the time axis (x-axis is assumed)
TimeAxis xAxis =   new TimeAxis(simpleTimeScale, xAxisTickMarkFormat);
xAxis.SetAxisTicksAttributes(xAxisMinorTickLength,
                      xAxisMajorTickLength, xAxisTickDirection);

// Create the linear y-axis
LinearAxis yAxis =
new LinearAxis(simpleTimeScale, ChartObj.Y_AXIS);

// Create the ChartView object to place graph objects in.
ChartView chartVu  = new ChartView();

// Add the x- and y-axes to the chartVu object
chartVu.AddChartObject(xAxis);
chartVu.AddChartObject(yAxis);
```

Elapsed Time Axes

Class ElapsedTimeAxis

GraphObj
|
+--Axis
|
+-- LinearAxis
|
+-- ElapsedTimeAxes

The **ElapsedTimeAxis** is subclassed from the **LinearAxis** class and has much in common with it. The only difference between the two is the way in which major and minor tick marks are calculated in the **CalcAutoAxis** method. The **ElapsedTimeAxis** takes into account the base 60 of seconds per minute and minutes per hour, and the base 24 of hours per day. Read the sections:

Linear Axis Minimum and Maximum

Linear Minor and Major Tick Mark Intervals

Linear Axis Intercept

Linear Axis Tick Mark Origin

under the discussion of **LinearAxis**.

Creating a Elapsed Time Axis

There are two main constructors for **ElapsedTimeAxes** objects. The first **ElapsedTimeAxes** constructor assumes that the axis extents match the extents of the underlying coordinate system, *transform*. The second **ElapsedTimeAxes** constructor sets the axis extents to the specified minimum and maximum values, regardless of the underlying coordinate system.

ElapsedTimeAxes constructors

```
[C#]
public ElapsedTimeAxis(
   PhysicalCoordinates transform,
   int axtype
);


public ElapsedTimeAxis (
   PhysicalCoordinates transform,
   int axtype,
   double minval,
   double maxval
);
```

transform	Places the axes in the coordinate system defined by transform.
axtype	Specifies if the axis is an x-axis (X_AXIS), or a y-axis (Y_AXIS).
minval	Sets the minimum value for the axis.
maxval	Sets the maximum value for the axis.

Other axis properties: minor tick mark spacing, number of minor tick marks per major tick mark, axis intercept, tick mark lengths, tick mark direction and axis tick mark origin are automatically calculated using an auto-axis method. Set these properties explicitly if you need to override the automatically calculated values.

SetAxisIntercept method

```
[C#]
public void SetAxisIntercept(
   double intercept
);
```

SetAxisTicks method

```
[C#]
public void SetAxisTicks(
   double tickorigin,
   double tickspace,
   int nticspermajor
);

public void SetAxisTicks(
   double tickorigin,
   double tickspace,
   int nminorticspermajor,
   double minorticklength,
   double majorticklength,
   int tickdir
);
```

intercept	Sets the intercept of this axis with the perpendicular axis in physical coordinates.
tickorigin	The tick marks start at this value.
tickspace	Specifies the spacing between minor tick marks.
nticspermajor	Specifies the number of minor tick marks per major tick mark.
minorticklength	The length of minor tick marks, in UWP device coordinates.
majorticklength	The length of major tick marks, in UWP device coordinates.
tickdir	The direction of the tick marks. Use one of the tick mark direction constants: AXIS_MIN, AXIS_CENTER, or AXIS_MAX.

Use the **SetLineWidth**, **SetLineStyle** and **SetColor** methods to customize the drawing properties of the lines used to draw the axis line and tick marks.

Simple elapsed time axis example

[C#]

```
// Define the coordinate system
TimeSpan xMin = TimeSpan.FromSeconds(0);
TimeSpan xMax = TimeSpan.FromSeconds(15);
double yMin = 0;
double yMax =  105;
ElapsedTimeCoordinates simpleScale =
```

```
            new ElapsedTimeCoordinates (xMin, yMin, xMax, yMax);

// Create the x- and y-axes
ElapsedTimeAxis xAxis = new ElapsedTimeAxis (simpleScale, ChartObj.X_AXIS);
ElapsedTimeAxis yAxis = new ElapsedTimeAxis (simpleScale, ChartObj.Y_AXIS);

// Add the x- and y-axes to the chartVu object
chartVu.AddChartObject(xAxis);
chartVu.AddChartObject(yAxis);
```

Event Axes

Class EventAxis

GraphObj
|
+--Axis
|
+-- LinearAxis
|
+--EventAxes

The **EventAxis** is subclassed from the **LinearAxis** class and has much in common with it. The major difference between the two is the way in which major and minor tick marks are calculated in the **CalcAutoAxis** method. The **EventAxis** places tick marks at the x-positions of the ChartEvent objects attached to the common EventCoordinate system. It places major tick marks at the ChartEvent objects which correspond to where an axis label is expected to go, and minor tick marks at events which fall between the major tick marks. If the tick marks start to overlap, tick marks are skipped in order to maintain the look of the chart.

Tick Rules

The TickRule property controls the tick mark logic of the axis. Use one of the TICK_RULE enumeration constants:

NO_TICKS – do not display any tick marks. No tick marks means no axis labels.

MINOREVENT_MAJOREVENT – display a minor tick mark every AxisMinorNthTick event, and a major tick mark every AxisMinorTicksPerMajor.

MAJOREVENT - display a major tick mark every AxisMajorNthTick event.

MINORCROSSOVEREVENT_MAJORCROSSOVEREVENT – display a minor tick mark every AxisMinorNthTick, minor crossover event, and a major tick mark every AxisMajorNthTick major crossover event. The minor and major crossover events are

controlled by the MajorTickCrossoverEvent and MinorTickCrossoverEvent properties of the EventAxis.

The default is ChartObj.TICK_RULE.MINORCROSSOVEREVENT_MAJORCROSSOVEREVENT.

The term crossover event means that an element of date/time timestamp changes. If you specify a MinorTickCrossoverEvent of ChartObj.SECOND, and an AxisMinorNthTick of 15, this will cause a minor tick mark to be displayed every 15th second, if an event falls within that range. So, if your events are spaced approximately 5 seconds apart, you will get a minor tick mark for approximately every three events. If you choose a MajorTickCrossoverEvent of ChartObj.MINUTE and an AxisMajorNthTick of 1, this will cause a major tick mark to be displayed every minute, if an event falls within that range. Tick marks only show up on an event, so if there are no events within the time interval, no tick mark will appear.

Creating a Event Axis

There are two main constructors for **EventAxis** objects. The first **EventAxis** constructor assumes that the axis extents match the extents of the underlying coordinate system, *transform*. The second **EventAxis** constructor sets the axis extents to the specified minimum and maximum values, regardless of the underlying coordinate system.

EventAxis constructors

By

```
[C#]
public ElapsedTimeAxis(
   PhysicalCoordinates transform,
   TICK_RULE tickrule,
   int axtype
);

public ElapsedTimeAxis (
   PhysicalCoordinates transform,
   TICK_RULE tickrule,
   int axtype,
   double minval,
   double maxval
);
```

transform Places the axes in the coordinate system defined by transform.

tickrule Specifies the tick rule used to calculated the tick marks: NO_TICKS,MINOREVENT_MAJOREVENT, MAJOREVENT, MINORCROSSOVEREVENT_MAJORCROSSOVEREVENT, MAJORCROSSOVEREVENT.

axtype Specifies if the axis is an x-axis (X_AXIS), or a y-axis (Y_AXIS).

minval Sets the minimum value for the axis.

maxval Sets the maximum value for the axis.

Other axis properties: minor tick mark spacing, number of minor tick marks per major tick mark, axis intercept, tick mark lengths, tick mark direction and axis tick mark origin are automatically calculated using an auto-axis method. Set these properties explicitly if you need to override the automatically calculated values.

SetAxisIntercept method

```
[C#]
public void SetAxisIntercept(
   double intercept
);
```

SetAxisTicks method

```
[C#]
public void SetAxisTicks(
   double tickorigin,
   double tickspace,
   int nticksperma jor
);

public void SetAxisTicks(
   double tickorigin,
   double tickspace,
   int nminorticksperma jor,
   double minorticklength,
   double majorticklength,
   int tickdir
);
```

intercept Sets the intercept of this axis with the perpendicular axis in physical coordinates.

tickorigin The tick marks start at this value.

tickspace Not used in event axis.

nticksperma jor Specifies the number of minor tick marks per major tick mark.

minorticklength The length of minor tick marks, in .Net device coordinates.

majorticklength The length of major tick marks, in .Net device coordinates.

tickdir The direction of the tick marks. Use one of the tick mark direction constants: AXIS_MIN, AXIS_CENTER, or AXIS_MAX.

Use the **SetLineWidth**, **SetLineStyle** and **SetColor** methods to customize the drawing properties of the lines used to draw the axis line and tick marks.

Simple event axis example

[C#]

```
EventSimpleDataset Dataset1 = new EventSimpleDataset("Actual Sales", chartevents);
 EventCoordinates pTransform1 = new EventCoordinates(Dataset1);
 pTransform1.SetScaleStartY(0);
 pTransform1.SetGraphBorderDiagonal(0.15, .15, .9, 0.8);
 Background background = new Background(pTransform1, ChartObj.GRAPH_BACKGROUND,
     Color.FromArgb(255,30, 70, 70), Color.FromArgb(255,90, 20, 155), ChartObj.Y_AXIS);
 chartVu.AddChartObject(background);

 EventAxis xAxis = new EventAxis(pTransform1, EventAxis.TICK_RULE.MAJOREVENT,
ChartObj.X_AXIS);
 xAxis.SetColor(Colors.White);
 chartVu.AddChartObject(xAxis);

 LinearAxis yAxis = new LinearAxis(pTransform1, ChartObj.Y_AXIS);
 yAxis.SetColor(Colors.White);
 chartVu.AddChartObject(yAxis);
```

Polar Axes

Polar axes provide the visual scale needed to compare data values that use polar coordinates. A polar axis consists of two parts. The first part is a pair of linear x- and y-axes intersecting at the center, Cartesian coordinate (0, 0). The second part of a polar axis is a circle with radius R, centered on the origin.

Class PolarAxes

GraphObj
|
+--Axis
|
+-- LinearAxis
|
+-- PolarAxes

The **PolarAxes** class creates a polar axes object that combines linear x- and y-axes for measurement of the polar magnitude, and a circular axis for measurement of the polar angle. The **PolarAxes** class extends the **LinearAxis** class. This is useful because the **LinearAxis** already has member variables that define properties and draw the tick marks for the circular axis. The **PolarAxes** class also includes uses two additional **LinearAxis** objects in support of the x and y linear axes used in the drawing of the polar magnitude axes.

Polar Axis Minimum and Maximum

Polar axes have only one scaling value, the maximum value of the polar magnitude, designated R. The minimum value is always 0.0. The limits of the x- and y-axis are set to +-R with the intercept for each axis set to 0.0. The polar angle scale is always 360 degrees, corresponding to a full circle.

Polar Axis Minor and Major Tick Mark Intervals

Polar axes use two sets of tick mark properties; one set for the x- and y-axes and the other set for the circular axis. The x- and y-axes use the same values for the major and minor tick mark spacing, partitioning the axes between +-R endpoints. The circular axis also uses major and minor tick marks, analogous to the hour and minute marks of a clock.

Creating polar axes

There is only one constructor for **PolarAxes** objects.

```
PolarAxes(PolarCoordinates transform)
```

transform The *transform* coordinate system defines the axis extents of the polar axes.

The **PolarAxes** constructor assumes that the axis extents match the extents of the underlying coordinate system, *transform*.

Other axis properties: minor tick mark spacing, number of minor tick marks per major tick mark, tick mark direction and tick mark lengths are automatically calculated using an auto-axis method. These properties can be explicitly set if you need to override the automatically calculated values.

```
[C#]
public void SetPolarAxesTicks(
   double axestickspace,
   int axesntickspermajor,
   double angletickspace,
   int anglentickspermajor
);

public void SetPolarAxesTicks(
   double axestickspace,
   int axesntickspermajor,
   double angletickspace,
   int anglentickspermajor,
   double minorticklength,
   double majorticklength,
   int tickdir
);
```

axestickspace	Specifies the spacing between minor tick marks for the x- and y-axes.
axesnticksperma jor	Specifies the number of minor tick marks per major tick mark for the x- and y-axes.
angletickspace	Specifies the spacing, in degrees, between minor tick marks for the radial axis.
anglenticksperma jor	Specifies the number of minor tick marks per major tick mark for the radial axis.
minorticlength	The length of minor tick marks, in UWP device coordinates.
majorticlength	The length of major tick marks, in UWP device coordinates.
tickdir	The direction of the tick marks. Use one of the tick mark direction constants: AXIS_MIN, AXIS_CENTER, or AXIS_MAX.

Use the **SetLineWidth**, **SetLineStyle** and **SetColor** methods to customize the drawing properties of the lines used to draw the axes lines and tick marks.

Simple polar axes example

[C#]

```
double polarmagnitude = 5.0;
PolarCoordinates polarscale = new PolarCoordinates(polarmagnitude);
PolarAxes polarAxes = new PolarAxes(polarscale);
// Create the ChartView object to place graph objects in.
ChartView chartVu  = new ChartView();
// Add the polar axes to the chartVu object
chartVu.AddChartObject(polarAxes);
```

Custom polar axes example

[C#]

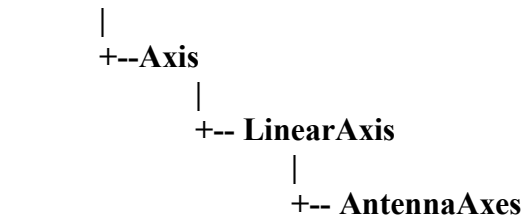
```
double polarmagnitude = 15.0;
PolarCoordinates polarscale = new PolarCoordinates(polarmagnitude);
PolarAxes polarAxes = new PolarAxes(polarscale);

double axestickspace = 1;
int axesnticksperma jor = 5;
double angletickspace = 50;
int anglenticksperma jor = 6;
double minorticlength = 5;
double majorticlength = 10;
int tickdir = ChartObj.AXIS_CENTER;

polarAxes.SetPolarAxesTicks(axestickspace, axesnticksperma jor,
               angletickspace, anglenticksperma jor,
               minorticlength, majorticlength,
```

```
                tickdir);
// Create the ChartView object to place graph objects in.
ChartView chartVu  = new ChartView();
// Add the polar axes to the chartVu object
chartVu.AddChartObject(polarAxes);
```

Antenna Axes

Antenna axes provide the visual scale needed to compare data values that use antenna coordinates. An antenna axis consists of two parts. The first part is a linear y-axis extending from the origin to the outer edge of the radial scale. The second part of an antenna axis is a circle with a radius equal to the range of the radial scale, centered on the origin.

Class AntennaAxes

GraphObj


The **AntennaAxes** class creates an antenna axes object that combines a linear y-axis for measurement of the radial values, and a circular axis for the measurement of the angular values. The **AntennaAxes** class extends the **LinearAxis** class. This is useful because the **LinearAxis** already has member variables that define properties and draw the tick marks for the circular axis.

Antenna Axis Minimum and Maximum

Antenna axes use two scaling values, a minimum and maximum radius value. The radius minimum value is set at the origin of the coordinate system, and the radius maximum value at the outer edge. The radius starting and ending values can be positive or negative. The maximum value should always be greater than the minimum value though. The angular scale is always 360 degrees, corresponding to a full circle. The angular scale starts with 0 degrees at 12:00 and increases clockwise.

Antenna Axis Minor and Major Tick Mark Intervals

Antenna axes use two sets of tick mark properties; one set for the y-axis and the other set for the circular axis. The y-axis has major and minor tick mark properties, as does the circular axis.

Creating antenna axes

There is only one constructor for **AntennaAxes** objects.

```
AntennaAxes(AntennaCoordinates transform)
```

transform The *transform* coordinate system defines the axis extents of the antenna axes.

The **AntennaAxes** constructor assumes that the axis extents match the extents of the underlying coordinate system, *transform*.

Other axis properties: minor tick mark spacing, number of minor tick marks per major tick mark, tick mark direction and tick mark lengths are automatically calculated using an auto-axis method. These properties can be explicitly set if you need to override the automatically calculated values.

```
[C#]
public void SetAntennaAxesTicks(
   double axestickspace,
   int axesntickspermajor,
   double angletickspace,
   int anglentickspermajor
);

public void SetAntennaAxesTicks(
   double axestickspace,
   int axesntickspermajor,
   double angletickspace,
   int anglentickspermajor,
   double minorticklength,
   double majorticklength,
   int tickdir
);
```

axestickspace Specifies the spacing between minor tick marks for the x- and y-axes.

axesntickspermajor Specifies the number of minor tick marks per major tick mark for the x- and y-axes.

angletickspace Specifies the spacing, in degrees, between minor tick marks for the radial axis.

anglentickspermajor Specifies the number of minor tick marks per major tick mark for the radial axis.

minorticlength The length of minor tick marks, in UWP device coordinates.

majorticlength The length of major tick marks, in UWP device coordinates.

tickdir The direction of the tick marks. Use one of the tick mark direction constants: AXIS_MIN, AXIS_CENTER, or AXIS_MAX.

Use the **SetLineWidth**, **SetLineStyle** and **SetColor** methods to customize the drawing properties of the lines used to draw the axes lines and tick marks.

Simple antenna axes example

[C#]

```
double minvalue  = -40, maxvalue = 20;
AntennaCoordinates antennascale = new AntennaCoordinates(minvalue, maxvalue);
AntennaAxes antennaAxes = new AntennaAxes(antennascale);
// Add the Antenna axes to the chartVu object
chartVu.AddChartObject(antennaAxes);
```

Custom antenna axes example

[C#]

```
double minvalue  = -40, maxvalue = 20;
AntennaCoordinates antennascale = new AntennaCoordinates(minvalue, maxvalue);
AntennaAxes antennaAxes = new AntennaAxes(antennascale);

double axestickspace = 1;
int axesnticksperma jor = 5;
double angletickspace = 5;
int anglenticksperma jor = 6;
double minorticlength = 5;
double majorticlength = 10;
int tickdir = ChartObj.AXIS_CENTER;

antennaAxes.SetAntennaAxesTicks(axestickspace, axesnticksperma jor,
               angletickspace, anglenticksperma jor,
               minorticlength, majorticlength,
               tickdir);
// Add the Antenna axes to the chartVu object
chartVu.AddChartObject(antennaAxes);
```

8. Axis Labels

AxisLabels
- **NumericAxisLabels**
- **TimeAxisLabels**
- **ElapsedTimeAxisLabels**
- **EventAxisLabels**
- **StringAxisLabels**
- **PolarAxesLabels**
- **AntennaAxesLabels**

Axis Labels

Axis labels are numeric or text strings placed next to axis tick marks, indicating the scale of the axis. Axis labels are a separate class from the axis classes. An axis class, i.e. any class derived from **Axis**, can exist independent of axis labels. Many graphs use axes that do not have labels. For example, the y-axis on the left side of a graph may have labels, while an identical axis on the right hand side of the graph may not. The axis label classes require a valid reference axis class and cannot exist independently.

There are seven axis labels classes: the **AxisLabels** abstract base class and concrete subclasses **NumericAxisLabels**, **TimeAxisLabels, ElapsedTimeAxisLabels, EventAxisLabels, StringAxisLabels**, **PolarAxesLabels** and **AntennaAxesLabels.**.

Label Formats

An axis label can take many forms. The various axis label formats are divided between the axis labels classes in the following manner.

NumericAxisLabels

The **LinearAxis** and **LogAxis** axis types use this class.

- Full decimal conversion (0.000015)
- Scientific notation (1.5e-5)
- Exponent notation ($1.5x10^{-5}$)
- Percent format (76%)
- Business format where B, M and K are used to represent billions, millions and thousands (1043M, 11.0K)
- Currency format ($123432)
- Business currency format – The business format combined with the currency format ($123K)

StringAxisLabels

The **LinearAxis** and **LogAxis** axis types use this class.

Arbitrary strings (“MA”, “PA”, “JEFF”,”Sales”) are used to label the major tick marks of the axis.

TimeAxisLabels

The **TimeAxis** axis types use this class.

- Time formats (hh:mm:ss, hh:mm, mm:ss, etc.)
- Date formats (mm/dd/yy, dd/mm/yy, mm/yy, etc.)
- Time/Date formats (mmm/ddd/yyy hh:mm:ss)

ElapsedTimeAxisLabels

The **ElapsedTimeAxisLabels** are used in combination with the **ElapsedTimeAxis** type**.**

Elapsed time formats (hh:mm:ss, hh:mm, mm:ss, mm:ss.fff, etc.)

EventAxisLabels

The EventeAxisLabels are used in combination with the **EventAxis** type**.**

- Time formats (hh:mm:ss, hh:mm, mm:ss, etc.)
- Date formats (mm/dd/yy, dd/mm/yy, mm/yy, etc.)
- Time/Date formats (mmm/ddd/yyy hh:mm:ss)

PolarAxesLabels

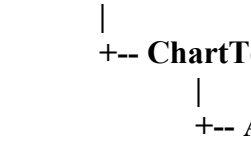
This class displays numeric labels exclusively for the **PolarAxes** class. It uses labels in the same format as the **NumericAxisLabels**.

AntennaAxesLabels

This class displays numeric labels exclusively for the **AntennaAxes** class. It uses labels in the same format as the **NumericAxisLabels**.

Class AxisLabels

GraphObj



The **AxisLabels** class is the abstract base class for all axis label objects. It contains the properties and methods common to subclasses implementing more specialized axis labels.

Axis Label Text

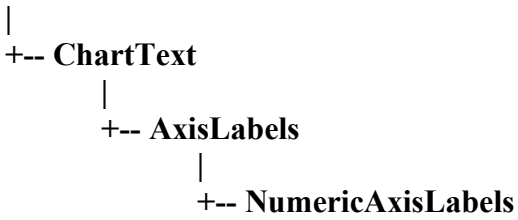
The **AxisLabels** class includes a reference to a **ChartFont** object. If a valid font is not supplied a default font is created and used. Every axis labels object can have a unique font associated with it. The **ChartFont** object defines the font typeface, size, and style. The font for any of the axis labels can be set using the **AxisLabels.SetTextFont** method. The **AxisLabels** class manages other text attributes not directly associated with the font. These include the text foreground color, the text background color and the rotation of the text if it is different from the normal horizontal orientation. It is common to rotate x-axis labels 90 degrees, so that they are vertical rather than horizontal, in order to squeeze more tick mark labels in along the x-axis.

Axis Labels Positioning

The **AxisLabels** class manages the placement of the axis labels with respect to the underlying axis tick marks. Labels can be place above or below the tick marks of a horizontal x-axis, and to the left or right of the tick marks for a vertical y-axis. The axis label justification constants AXIS_MIN and AXIS_MAX are used for this purpose. The AXIS_MIN constant places the text label on the side of the tick mark that is in the direction of the perpendicular axis coordinate system minimum. The AXIS_MAX is much the same, except that it places the label on the side that is in the direction of the perpendicular axis coordinate system maximum. Axis labels should not actually touch the tick marks, so x and y offsets are factored in. The programmer can modify these offsets.

Numeric Axis Labels

Class NumericAxisLabels

GraphObj


The **NumericAxisLabels** class extends the **AxisLabels** class, adding extensive numeric formatting capability. It labels axes created using the **LinearAxis** and **LogAxis** classes.

Label formats

An axis label can take many forms. Variations on these forms include:

- Full decimal conversion (0.000015)
- Scientific notation (1.5e-5)
- Exponent notation ($1.5x10^{-5}$)
- Percent format (76%)
- Business format where B, M and K are used to represent billions, millions and thousands (1043M, 11.0K)
- Currency format ($123432)
- Business currency format – The business format combined with the currency format ($123K)

Depending on the scaling of the associated axis, the numeric values of the axis labels may be very large or very small numbers requiring a great deal of space to display. Various techniques are used to abbreviate the numeric value, reducing the space requirements. Expressing a numeric value in scientific notation can reduce the amount of space a label requires, if the numeric value requires eight or more digits. If the label values end in a lot of zeros (10000000, 9000000, 8000000…), a major reduction in space is achieved by using the business format which replaces all of the zeros with a letter (10M, 9M, 8M, …).

The axis numeric labels constants are listed below:

Numeric Format Constant	**Description**
DECIMALFORMAT	Decimal format, i.e. 1234.563
SCIENTIFICFORMAT	Scientific or exponential format: 1.23e3
BUSINESSFORMAT	Business format where the letters K, M, B or T are appended on the end a truncated numeric value, i.e. 1.23, 14K, 44M, 32B, 3.0T
ENGINEERINGFORMAT	If the absolute value of the label is greater than 1.0e6, or less than 1.0e-6, the scientific format is used, else the decimal format is used.
PERCENTFORMAT	The value of a label is multiplied by 100 and the character '%' is appended on the end of the label.
EXPONENTFORMAT	The value of a label is multiplied by 100 and the character '%' is appended on the end of the label.
CURRENCEYBUSINESSFORMAT	A '$' character is appended to the front of the label and the values are abbreviated the same as the BUSINESSFORMAT
CURRENCEYFORMAT	A '$' character is appended to the front of the label. .

NumericAxisLabels constructor

There is only one main constructor for **NumericAxisLabels** objects.

```
[C#]
public NumericAxisLabels(
   Axis baseaxis
);
```

baseaxis This is the axis the axis labels are for.

Other axis label properties: font, rotation, numeric format, axis labels direction and numeric precision are automatically set. These properties can be explicitly set if you need to override the automatically calculated values.

SetAxisLabels method

```
[C#]
public void SetAxisLabels(
   ChartFont font,
   double rotation,
   int labdir,
   int decimalpos,
   int labelends,
   Color labcolor
);
```

SetAxisLabelsFormat method

```
[C#]
public void SetAxisLabelsFormat(
   int format
);
```

font The font object used to display the axis label text.

rotation The rotation, in degrees, of label text in the normal viewing plane.

labdir The justification of the axis label (AXIS_MIN or AXIS_MAX) with respect to the tick mark endpoint.

decimal Sets the number of digits to the right of the decimal point for numeric axis labels.

labelends Specifies whether there should be labels for the axis minimum (LABEL_MIN), maximum (LABEL_MAX) or tick mark starting point (LABEL_ORIGIN). The value of these constants can be OR'd together. The value of LABEL_MIN | LABEL_MAX | LABEL_ORIGIN is LABEL_ALL

labcolor The color of the label text.

format Sets the numeric format for the axis labels. Use one of the numeric format constants: DECIMALFORMAT, SCIENTIFICFORMAT, EXPONENTFORMAT, BUSINESSFORMAT, ENGINEERINGFORMAT, PERCENTFORMAT, CURRENCEYFORMAT, CURRENCYBUSINESSFORMAT.

Simple numeric axis labels example

[C#]

```
// Define the coordinate system
double xMin = -5;
double xMax = 15;
double yMin = 0;
double yMax =  105;
CartesianCoordinates simpleScale =
               new CartesianCoordinates(xMin, yMin, xMax, yMax);

// Create the x- and y-axes
LinearAxis xAxis =
new LinearAxis(simpleScale, ChartObj.X_AXIS);
LinearAxis yAxis = new LinearAxis(simpleScale, ChartObj.Y_AXIS);
NumericAxisLabels xAxisLabels =  new NumericAxisLabels(xAxis);
NumericAxisLabels yAxisLabels =  new NumericAxisLabels(yAxis);
// Create the ChartView object to place graph objects in.
ChartView chartVu  = new ChartView();

// Add the x- and y-axes to the chartVu object
chartVu.AddChartObject(xAxis);
chartVu.AddChartObject(yAxis);
chartVu.AddChartObject(xAxisLabels);
chartVu.AddChartObject(yAxisLabels);
```

Should want to customize the axis you can add the following lines after the *xAxisLabels* object is created:

Custom numeric axis labels example

[C#]

```
ChartFont labelfont = new ChartFont("Microsoft Sans Serif", 10, FontStyle.Normal,
FontWeights.Bold);

double xAxisLabelsRotation = 0.0;
int xAxisLabelsDir = ChartObj.AXIS_MIN;
int xAxisLabelsDecimal = 1;
int xAxisLabelsEnds = ChartObj.LABEL_ALL;
Color xAxisLabelsColor = Colors.Black;
int xAxisNumericFormat = ChartObj.DECIMALFORMAT;

xAxisLabels.SetAxisLabels( labelfont, xAxisLabelsRotation,
                           xAxisLabelsDir, xAxisLabelsDecimal,
                           xAxisLabelsEnds, xAxisLabelsColor);
xAxisLabels.SetAxisLabelsFormat(xAxisNumericFormat);
```

String Axis Labels

Class StringAxisLabels

GraphObj
|
+-- ChartText
|
+-- AxisLabels
|
+-- StringAxisLabels

Use the **StringAxisLabels** class to label major tick marks of a linear or logarithmic axis with arbitrary strings.

StringAxisLabels constructor

There is only one main constructor for **StringAxisLabels** objects.

```
[C#]
public StringAxisLabels(
   Axis baseaxis
);
```

baseaxis This is the axis the axis labels are for.

The axis strings, and other axis label properties: font, rotation, numeric format, axis labels direction and numeric precision are automatically set. These properties can be explicitly set if you need to override the automatically calculated values.

SetAxisLabels method

```
C#
public void SetAxisLabels(
    ChartFont font,
    double rotation,
    int labdir,
    int labelends,
    Color labcolor,
    string[] tickstrings,
    int numtickstrings
)
```

font The font object used to display the axis label text.

rotation	The rotation, in degrees, of label text in the normal viewing plane.
labdir	The justification of the axis label (AXIS_MIN or AXIS_MAX) with respect to the tick mark endpoint.
decimal	Sets the number of digits to the right of the decimal point for numeric axis labels.
labelends	Specifies whether there should be labels for the axis minimum (LABEL_MIN), maximum (LABEL_MAX) or tick mark starting point (LABEL_ORIGIN). The value of these constants can be OR'd together. The value of LABEL_MIN | LABEL_MAX | LABEL_ORIGIN is LABEL_ALL
labcolor	The color of the label text.
tickstrings	*A*rray of strings, one for each major tick mark that you want labeled.
numtickstrings	The number of strings in the tickstrings array.

If you want the first major tick mark, usually the intercept of the y-axis with the x-axis, to remain empty, just initialize the .first element of the tickstrings array to the empty string, “”, as in the example below.

Simple string axis labels example, extracted from the BarGraphs.LandOfTheFry example program.

[C#]

```
int NumberOfCountries = 13;
int NumberOfGroups = 2;
String [] CountryNames =
    {
    "","China", "Japan", "Russia", "Italy", "Sweden",
    "Denmark", "Mexico", "Brazil", "France",
    "Australia", "Spain", "United States", "England"};

    // Positions bar
double []x1 = new double[NumberOfCountries];
.
.
.
// Y-axis string labels
// Each string will label a major tick mark
StringAxisLabels yAxisLab1 = new StringAxisLabels(yAxis1);
yAxisLab1.SetAxisLabels(theFont,0,
    ChartObj.AXIS_MIN, ChartObj.LABEL_ALL,
    Colors.Black,CountryNames,14);
    yAxisLab1.SetColor(Colors.Black);
        chartVu.AddChartObject(yAxisLab1);
```

Time and Date Axis Labels

Class TimeAxisLabels

GraphObj
|
+-- ChartText
|
+-- AxisLabels
|
+-- TimeAxisLabels

The **TimeAxisLabels** class extends the **AxisLabels** class, adding extensive time and date formatting capability. Use it to label axes created using the **TimeAxis** class.

Label formats

A time axis label can take many forms. Variations on these forms include:

- Time formats (hh:mm:ss, hh:mm, mm:ss, etc.)
- Date formats (mm/dd/yy, dd/mm/yy, mm/yy, etc.)

There are more ways to format time and date information than numeric data. The **QCChart2D for Windows Apps** software directly supports twelve time formats and eighteen date formats. It is also possible to create custom date/time formats. The software makes use of the **System.DateTime.ToString** `method` to format times and dates. A table listing predefined date/time formats appears below.

Date/Time Format Constant	Format String	Example String Result
TIMEDATEFORMAT_MSDDD	"mm:ss.fff"	12.33.999
TIMEDATEFORMAT_MSDD	"mm:ss.ff"	12.33.99
TIMEDATEFORMAT_MSD	"mm:ss.f"	12.33.9
TIMEDATEFORMAT_MS	"m:ss"	12:33
TIMEDATEFORMAT_12HMSDD	"h:mm.ss.ff"	11:12:33.99
TIMEDATEFORMAT_12HMSD	"h:mm.ss.f"	11:12:33.9
TIMEDATEFORMAT_12HMS	"h:mm:ss"	11:12:33
TIMEDATEFORMAT_12HM	"h:mm"	11:12

TIMEDATEFORMAT_24HMDDD	"H:mm:ss.ff"	23:12:33.99
TIMEDATEFORMAT_24HMDD	"H:mm:ss.f"	23:12:33.9
TIMEDATEFORMAT_24HMS	"H:mm:ss"	23:12:33
TIMEDATEFORMAT_24HM	"H:mm"	23:12
TIMEDATEFORMAT_STANDARD	"MMMMM dd, yyyy"	December 7, 2000
TIMEDATEFORMAT_MDY	"M/dd/yy"	12/07/00
TIMEDATEFORMAT_DMY	"d/MM/yy"	7/12/00
TIMEDATEFORMAT_MY	"M/yy"	7/00
TIMEDATEFORMAT_Q	None	Q1
TIMEDATEFORMAT_MMMM	"MMMM"	January
TIMEDATEFORMAT_MMM	"MMM"	Jan
TIMEDATEFORMAT_M	"MMM"	J
TIMEDATEFORMAT_DDDD	"dddd"	Tuesday
TIMEDATEFORMAT_DDD	"ddd"	Tue
TIMEDATEFORMAT_D	"ddd"	T
TIMEDATEFORMAT_Y	"yy"	00
TIMEDATEFORMAT_MDY2000	"M/dd/yyyy"	12/07/2000
TIMEDATEFORMAT_DMY2000	"d/MM/yyyy"	7/12/2000
TIMEDATEFORMAT_MY2000	"M/yyyy"	7/2000
TIMEDATEFORMAT_Y2000	"yyyy"	2000
TIMEDATEFORMAT_MDY_HMS	"H:mm:ss\nM/dd/yy"	12:23:33 12/07/00
TIMEDATEFORMAT_DMY_HMS	"H:mm:ss\nd/M/yy"	23:12:33 12/07/00

TIMEDATEFORMAT_MDY_HM	"H:mm\nM/dd/yy"	12:23:33 12/07/00
TIMEDATEFORMAT_DMY_HM	"H:mm\nd/M/yy"	23:12:33 12/07/00

In some cases, the TIMEDATEFORMAT_Q format for example, the **DateTime.ToString** class does not handle the desired conversion. In cases like this the date/time Format constant is trapped and undergoes additional processing to create the final label. That is why some of the date format strings are the same, event though the resulting labels are different.

TimeAxis Labels constructor

There is only one main constructor for **TimeAxisLabels** objects.

```
[C#]
public TimeAxisLabels(
   TimeAxis baseaxis
);
```

baseaxis This is the time axis the axis labels are for.

Other axis label properties: font, rotation, numeric format, axis labels direction and numeric precision are automatically set. These properties can be explicitly set if you need to override the automatically calculated values.

SetAxisLabels method

```
[C#]
public void SetAxisLabels(
   ChartFont font,
   double rotation,
   int labdir,
   int decimalpos,
   int labelends,
   Color labcolor
);
```

SetAxisLabelsFormat method

```
[C#]
public void SetAxisLabelsFormat(
   int format
);
```

font	The font object used to display the axis label text.
rotation	The rotation, in degrees, of label text in the normal viewing plane.
labdir	The justification of the axis label (AXIS_MIN or AXIS_MAX) with respect to the tick mark endpoint.
decimal	Sets the number of digits to the right of the decimal point for numeric axis labels.
labelends	Ignored for time axis labels
labcolor	The color of the label text.
format	Sets the numeric format for the axis labels. Use one of the time format constants: TIMEDATEFORMAT_MSDDD, TIMEDATEFORMAT_MSDD, TIMEDATEFORMAT_MSD, TIMEDATEFORMAT_MS, TIMEDATEFORMAT_12HMSDD, TIMEDATEFORMAT_12HMSD, TIMEDATEFORMAT_12HMS, TIMEDATEFORMAT_12HM, TIMEDATEFORMAT_24HMSDD, TIMEDATEFORMAT_24HMSD, TIMEDATEFORMAT_24HMS, TIMEDATEFORMAT_24HM, TIMEDATEFORMAT_STANDARD, TIMEDATEFORMAT_MDY,TIMEDATEFORMAT_DMY, TIMEDATEFORMAT_MY, TIMEDATEFORMAT_Q, TIMEDATEFORMAT_MMMM, TIMEDATEFORMAT_MMM, TIMEDATEFORMAT_M, TIMEDATEFORMAT_DDDD, TIMEDATEFORMAT_DDD, TIMEDATEFORMAT_D, TIMEDATEFORMAT_Y, TIMEDATEFORMAT_MDY2000, TIMEDATEFORMAT_DMY2000, TIMEDATEFORMAT_MY2000, TIMEDATEFORMAT_Y2000, TIMEDATEFORMAT_MDY_HMS, TIMEDATEFORMAT_DMY_HMS, TIMEDATEFORMAT_MDY_HM, TIMEDATEFORMAT_DMY_HM.

Simple time axis labels example

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
// Define a Time coordinate system
ChartCalendar xMin = new ChartCalendar(1996, ChartObj.FEBRUARY,  5);
ChartCalendar xMax = new ChartCalendar(2002, ChartObj.JANUARY,  5);
```

```
double yMin = 0;
double yMax =  105;

TimeCoordinates simpleTimeScale;
simpleTimeScale = new TimeCoordinates(xMin, yMin, xMax, yMax);
// Create the time axis (x-axis is assumed)
TimeAxis xAxis = new TimeAxis(simpleTimeScale);
// Create the linear y-axis
LinearAxis yAxis =
new LinearAxis(simpleTimeScale, ChartObj.Y_AXIS);
TimeAxisLabels xAxisLabels = new TimeAxisLabels(xAxis);
NumericAxisLabels yAxisLabels = new NumericAxisLabels(yAxis);


// Add the x- and y-axes to the chartVu object
chartVu.AddChartObject(xAxis);
chartVu.AddChartObject(yAxis);
chartVu.AddChartObject(xAxisLabels);
chartVu.AddChartObject(yAxisLabels);
```

Custom time axis labels example

[C#]

```
ChartFont labelfont = new ChartFont("Microsoft Sans Serif", 10, FontStyle.Normal,
FontWeights.Bold);

double xAxisLabelsRotation = 0.0;
int xAxisLabelsDir = ChartObj.AXIS_MIN;
int xAxisLabelsEnds = ChartObj.LABEL_ALL;
Color xAxisLabelsColor = Colors.Black;
int xAxisNumericFormat = ChartObj.TIMEDATEFORMAT_MY;

xAxisLabels.SetAxisLabels( labelfont, xAxisLabelsRotation,
                          xAxisLabelsDir,
                          xAxisLabelsEnds, xAxisLabelsColor);
xAxisLabels.SetAxisLabelsFormat(xAxisNumericFormat);
```

Elapsed Time Axis Labels

Class ElapsedTimeAxisLabels

GraphObj
|
+-- ChartText
|
+-- AxisLabels
|
+-- ElapsedTimeAxisLabels

The **ElapsedTimeAxisLabels** class extends the **AxisLabels** class and provides for elapsed time labels. It adds extensive time formatting capability. Use it to label axes created using the **ElapsedTimeAxis** class.

Elapsed Time Label formats

A time axis label can take several forms. The TimeFormat property controls the elapsed time format.

TimeFormat Format Constant	Example String Result
TIMEDATEFORMAT_MS	12:33
TIMEDATEFORMAT_24HMS	23:12:33
TIMEDATEFORMAT_24HM	23:12

The TIMEDATEFORMAT_MS and TIMEDATEFORMAT formats can also have a decimal precision, appending a decimal point and the specified number of significant digits to the right of the time lable seconds, i.e. 12:33.7432. This is set using the **AxisLabelsDecimalPos** property.

ElapsedTimeAxis Labels constructor

There is only one main constructor for **TimeAxisLabels** objects.

```
[C#]
public ElapsedTimeAxisLabels(
   ElapsedTimeAxis baseaxis
);
```

baseaxis This is the elapsed time axis the axis labels are for.

Other axis label properties: font, rotation, time format, axis labels direction and numeric precision are automatically set. These properties can be explicitly set if you need to override the automatically calculated values.

SetAxisLabels method

```
[C#]
public void SetAxisLabels(
   ChartFont font,
   double rotation,
   int labdir,
   int decimalpos,
   int timeformat,
   int labelends,
   Color labcolor
);
```

SetAxisLabelsFormat method

```
[C#]
public void SetAxisLabelsFormat(
   int format
);
```

font	The font object used to display the axis label text.
rotation	The rotation, in degrees, of label text in the normal viewing plane.
labdir	The justification of the axis label (AXIS_MIN or AXIS_MAX) with respect to the tick mark endpoint.
decimal	Sets the number of digits to the right of the decimal point for elapsed time axis labels.
labelends	Ignored for time axis labels
labcolor	The color of the label text.
timeformat	Sets the numeric format for the axis labels. Use one of the time format constants: TIMEDATEFORMAT_MS, TIMEDATEFORMAT_24HMS, TIMEDATEFORMAT_24HM.

Simple ElapsedTimeAxisLabels example (extracted from the NewDemosRev2.ElapsedTimeChart example program)

[C#]

```
TimeSpan[] x1 = new TimeSpan[numPoints];
double []y1 = new double[numPoints];
double []y2 = new double[numPoints];
int i;

for (i=0; i < numPoints; i++)
{
  // Initialize Data
  .
  .
  .
}

theFont = new ChartFont("Microsoft Sans Serif", 10,  FontStyle.Normal, FontWeights.Bold);
ElapsedTimeSimpleDataset Dataset1 = new ElapsedTimeSimpleDataset("First", x1, y1);
ElapsedTimeSimpleDataset Dataset2 =
    new ElapsedTimeSimpleDataset("Second", x1, y2);


ElapsedTimeCoordinates pTransform1 =
```

```
    new ElapsedTimeCoordinates(ChartObj.ELAPSEDTIME_SCALE, ChartObj.LINEAR_SCALE);
pTransform1.ElapsedTimeAutoScale(Dataset2, ChartObj.X_AXIS,
    ChartObj.AUTOAXES_FAR, ChartObj.AUTOAXES_FAR);
pTransform1.SetGraphBorderDiagonal(0.15, .15, .90, 0.70) ;
.
.
.
ElapsedTimeAxis xAxis = new ElapsedTimeAxis(pTransform1, ChartObj.X_AXIS);
chartVu.AddChartObject(xAxis);

LinearAxis yAxis = new LinearAxis(pTransform1, ChartObj.Y_AXIS);
chartVu.AddChartObject(yAxis);

ElapsedTimeAxisLabels xAxisLab = new ElapsedTimeAxisLabels(xAxis);
xAxisLab.SetTextFont(theFont);
xAxisLab.AxisLabelsDayFormat = ChartObj.ELAPSEDTIMEFORMAT_NEXTTODAYSTRING;
chartVu.AddChartObject(xAxisLab);

NumericAxisLabels yAxisLab = new NumericAxisLabels(yAxis);
yAxisLab.SetTextFont(theFont);
chartVu.AddChartObject(yAxisLab);
```

Event Axis Labels

Class EventAxisLabels

GraphObj
|
+-- ChartText
|
+-- AxisLabels
|
+-- EventAxisLabels

The **EventAxisLabels** class extends the **AxisLabels** class, adding extensive time and date formatting capability. Use it to label axes created using the **EventAxis** class.

Label formats

A time axis label can take many forms. Variations on these forms include:

- Time formats (hh:mm:ss, hh:mm, mm:ss, etc.)
- Date formats (mm/dd/yy, dd/mm/yy, mm/yy, etc.)

There are more ways to format time and date information than numeric data. The **QCChart2D for *.Net*** software directly supports twelve time formats and twenty-two date formats. It is also possible to create custom date/time formats. The software makes use of the **System.DateTime.ToString** `method` to format times and dates. A table listing predefined date/time formats appears below.

Date/Time Format Constant	Format String	Example String Result
TIMEDATEFORMAT_MSDDD	"mm:ss.fff"	12.33.999
TIMEDATEFORMAT_MSDD	"mm:ss.ff"	12.33.99
TIMEDATEFORMAT_MSD	"mm:ss.f"	12.33.9
TIMEDATEFORMAT_MS	"m:ss"	12:33
TIMEDATEFORMAT_12HMSDD	"h:mm.ss.ff"	11:12:33.99
TIMEDATEFORMAT_12HMSD	"h:mm.ss.f"	11:12:33.9
TIMEDATEFORMAT_12HMS	"h:mm:ss"	11:12:33
TIMEDATEFORMAT_12HM	"h:mm"	11:12
TIMEDATEFORMAT_24HMDDD	"H:mm:ss.ff"	23:12:33.99
TIMEDATEFORMAT_24HMDD	"H:mm:ss.f"	23:12:33.9
TIMEDATEFORMAT_24HMS	"H:mm:ss"	23:12:33
TIMEDATEFORMAT_24HM	"H:mm"	23:12
TIMEDATEFORMAT_STANDARD	"MMMMM dd, yyyy"	December 7, 2000
TIMEDATEFORMAT_MDY	"M/dd/yy"	12/07/00
TIMEDATEFORMAT_DMY	"d/MM/yy"	7/12/00
TIMEDATEFORMAT_MY	"M/yy"	7/00
TIMEDATEFORMAT_Q	None	Q1
TIMEDATEFORMAT_MMMM	"MMMM"	January
TIMEDATEFORMAT_MMM	"MMM"	Jan
TIMEDATEFORMAT_M	"MMM"	J
TIMEDATEFORMAT_DDDD	"dddd"	Tuesday
TIMEDATEFORMAT_DDD	"ddd"	Tue
TIMEDATEFORMAT_D	"ddd"	T

TIMEDATEFORMAT_Y	"yy"	00
TIMEDATEFORMAT_MDY2000	"M/dd/yyyy"	12/07/2000
TIMEDATEFORMAT_DMY2000	"d/MM/yyyy"	7/12/2000
TIMEDATEFORMAT_MY2000	"M/yyyy"	7/2000
TIMEDATEFORMAT_Y2000	"yyyy"	2000
TIMEDATEFORMAT_MDY_HMS	"H:mm:ss\nM/dd/yy"	12:23:33 12/07/00
TIMEDATEFORMAT_DMY_HMS	"H:mm:ss\nd/M/yy"	23:12:33 12/07/00
TIMEDATEFORMAT_MDY_HM	"H:mm\nM/dd/yy"	12:23:33 12/07/00
TIMEDATEFORMAT_DMY_HM	"H:mm\nd/M/yy"	23:12:33 12/07/00

In some cases, the TIMEDATEFORMAT_Q format for example, the **DateTime.ToString** class does not handle the desired conversion. In cases like this the date/time Format constant is trapped and undergoes additional processing to create the final label. That is why some of the date format strings are the same, event though the resulting labels are different.

EventAxis Labels constructor

There is only one main constructor for **EventAxisLabels** objects.

```
[C#]
public TimeAxisLabels(
   EventAxis baseaxis
);
```

baseaxis This is the event axis the axis labels are for.

Other axis label properties: font, rotation, numeric format, axis labels direction and numeric precision are automatically set. These properties can be explicitly set if you need to override the automatically calculated values.

SetAxisLabels method

```
[C#]
public void SetAxisLabels(
   Font font,
   double rotation,
   int labdir,
   int decimalpos,
   int labelends,
   Color labcolor
);
```

SetAxisLabelsFormat method

```
[C#]
public void SetAxisLabelsFormat(
   int format
);
```

font The font object used to display the axis label text.

rotation The rotation, in degrees, of label text in the normal viewing plane.

labdir The justification of the axis label (AXIS_MIN or AXIS_MAX) with respect to the tick mark endpoint.

decimal Sets the number of digits to the right of the decimal point for numeric axis labels.

labelends Ignored for time axis labels

labcolor The color of the label text.

format Sets the numeric format for the axis labels. Use one of the time format constants:

TIMEDATEFORMAT_MSDDD, TIMEDATEFORMAT_MSDD,
TIMEDATEFORMAT_MSD, TIMEDATEFORMAT_MS,
TIMEDATEFORMAT_12HMSDD, TIMEDATEFORMAT_12HMSD,
TIMEDATEFORMAT_12HMS, TIMEDATEFORMAT_12HM,
TIMEDATEFORMAT_24HMSDD, TIMEDATEFORMAT_24HMSD,
TIMEDATEFORMAT_24HMS, TIMEDATEFORMAT_24HM,
TIMEDATEFORMAT_STANDARD,
TIMEDATEFORMAT_MDY,TIMEDATEFORMAT_DMY,
TIMEDATEFORMAT_MY, TIMEDATEFORMAT_Q,
TIMEDATEFORMAT_MMMM, TIMEDATEFORMAT_MMM,
TIMEDATEFORMAT_M, TIMEDATEFORMAT_DDDD,
TIMEDATEFORMAT_DDD, TIMEDATEFORMAT_D,
TIMEDATEFORMAT_Y, TIMEDATEFORMAT_MDY2000,
TIMEDATEFORMAT_DMY2000, TIMEDATEFORMAT_MY2000,
TIMEDATEFORMAT_Y2000, TIMEDATEFORMAT_MDY_HMS,

TIMEDATEFORMAT_DMY_HMS,TIMEDATEFORMAT_MDY_HM
,TIMEDATEFORMAT_DMY_HMS .

Simple time axis labels example

[C#]

```
theFont = new ChartFont("Microsoft Sans Serif", 10, FontStyle.Normal);

EventSimpleDataset Dataset1 = new EventSimpleDataset("Actual Sales", chartevents);
EventCoordinates pTransform1 = new EventCoordinates(Dataset1);
pTransform1.SetScaleStartY(0);

EventAxis xAxis = new EventAxis(pTransform1, EventAxis.TICK_RULE.MAJOREVENT,
ChartObj.X_AXIS);
xAxis.SetColor(Colors.White);
chartVu.AddChartObject(xAxis);

LinearAxis yAxis = new LinearAxis(pTransform1, ChartObj.Y_AXIS);
yAxis.SetColor(Colors.White);
chartVu.AddChartObject(yAxis);

EventAxisLabels xAxisLab = new EventAxisLabels(xAxis);
xAxisLab.SetAxisLabelsFormat(ChartObj.TIMEDATEFORMAT_Y2000);
xAxisLab.SetColor(Colors.White);
chartVu.AddChartObject(xAxisLab);
```

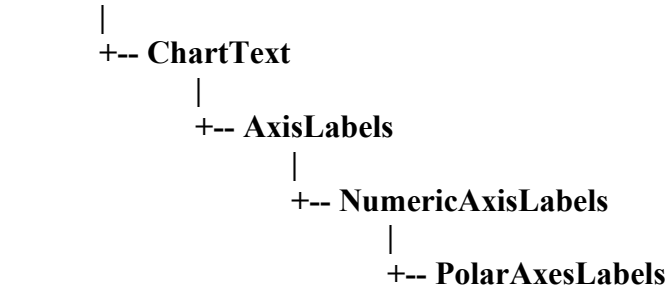
Custom time axis labels example

[C#]

```
ChartFont labelfont = new ChartFont("Microsoft Sans Serif", 10, FontStyle.Normal);

double xAxisLabelsRotation = 0.0;
int xAxisLabelsDir = ChartObj.AXIS_MIN;
int xAxisLabelsEnds = ChartObj.LABEL_ALL;
Color xAxisLabelsColor = Color.Black;
int xAxisNumericFormat = ChartObj.TIMEDATEFORMAT_MY;

xAxisLabels.SetAxisLabels( labelfont, xAxisLabelsRotation,
                          xAxisLabelsDir,
                          xAxisLabelsEnds, xAxisLabelsColor);
xAxisLabels.SetAxisLabelsFormat(xAxisNumericFormat);
```

Polar Axes Labels

Class PolarAxesLabels

GraphObj


The **PolarAxesLabels** class extends the **NumericAxisLabels** class and creates labels for objects of the **PolarAxes** class. The **PolarAxesLabels** class labels the two parts of the polar axes: the x- and y-axes pair defining the polar magnitude, and the polar angle circle, bounding the x- and y-axes. The class extends the **NumericAxisLabels** class and uses that class's methods and properties for managing the label properties.

The x- and y-axes have extents of +-R. The only labels needed for these axes are for the positive section of the x-axis. The easiest way to manage this is to create a local x-axis that extends from 0 to +R. This local axis is not drawn, but is used to create a **NumericAxisLabels** object for the class. This object draws the labels for the positive section of the x-axis.

PolarAxisLabels constructor

There is only one main constructor for **PolarAxesLabels** objects.

```
[C#]
public PolarAxesLabels(
   PolarAxes baseaxis
);
```

baseaxis This is the axis the axis labels are for.

Other axis label properties: font, rotation, numeric format, axis labels direction and numeric precision are automatically set. These properties can be explicitly set if you need to override the automatically calculated values.

SetAxisLabels method

```
[C#]
public void SetAxisLabels(
   ChartFont font,
   Color labcolor
);
```

SetAxisLabelsFormat method

```
[C#]
public void SetAxisLabelsFormat(
   int format
);
```

font	The font object used to display the axis label text.
labcolor	The color of the label text.
format	Sets the numeric format for the axis labels. Use one of the numeric format constants: DECIMALFORMAT, SCIENTIFICFORMAT, EXPONENTFORMAT, BUSINESSFORMAT, ENGINEERINGFORMAT, PERCENTFORMAT, CURRENCEYFORMAT, CURRENCYBUSINESSFORMAT.

Polar axes labels example

[C#]

```
double polarmagnitude = 5.0;
PolarCoordinates polarscale = new PolarCoordinates(polarmagnitude);
PolarAxes polarAxes = new PolarAxes(polarscale);

PolarAxesLabels polarAxesLabels = new PolarAxesLabels(polarAxes);
polarAxesLabels.SetAxisLabelsFormat(ChartObj.DECIMALFORMAT);
polarAxesLabels.SetAxisLabelsDecimalPos(2);

// Create the ChartView object to place graph objects in.
ChartView chartVu  = new ChartView();
// Add the polar axes to the chartVu object
chartVu.AddChartObject(polarAxes);
chartVu.AddChartObject(polarAxesLabels);
```

Antenna Axes Labels

Class AntennaAxesLabels

GraphObj
|
+-- ChartText
|
+-- AxisLabels
|
+-- NumericAxisLabels
|

+-- AntennaAxesLabels

The **AntennaAxesLabels** class extends the **NumericAxisLabels** class and creates labels for objects of the **AntennaAxes** class. The **AntennaAxesLabels** class labels the two parts of the antenna axes: the y-axis displaying the radius limits, and the angular circle bounding the y-axis. The class extends the **NumericAxisLabels** class and uses that class's methods and properties for managing the label properties.

AntennaAxisLabels constructor

There is only one main constructor for **AntennaAxesLabels** objects.

```
[C#]
public AntennaAxesLabels(
   AntennaAxes baseaxis
);
```

baseaxis This is the axis the axis labels are for.

Other axis label properties: font, rotation, numeric format, axis labels direction and numeric precision are automatically set. These properties can be explicitly set if you need to override the automatically calculated values.

SetAxisLabels method

```
[C#]
public void SetAxisLabels(
   ChartFont font,
   Color labcolor
);
```

SetAxisLabelsFormat method

```
[C#]
public void SetAxisLabelsFormat(
   int format
);
```

font The font object used to display the axis label text.

labcolor The color of the label text.

format Sets the numeric format for the axis labels. Use one of the numeric format constants: DECIMALFORMAT, SCIENTIFICFORMAT, EXPONENTFORMAT, BUSINESSFORMAT, ENGINEERINGFORMAT, PERCENTFORMAT, CURRENCEYFORMAT, CURRENCYBUSINESSFORMAT.

Antenna axes labels example

[C#]

```
double minvalue  = -40, maxvalue = 20;
AntennaCoordinates antennascale = new AntennaCoordinates(minvalue, maxvalue);
AntennaAxes antennaAxes = new AntennaAxes(antennascale);

chartVu.AddChartObject(antennaAxes);

AntennaAxesLabels antennaAxesLabels = new AntennaAxesLabels (antennaAxes);
antennaAxesLabels.SetAxisLabelsFormat(ChartObj.DECIMALFORMAT);
antennaAxesLabels.SetAxisLabelsDecimalPos(2);

chartVu.AddChartObject(antennaAxesLabels);
```

9. Axis Grids

ChartGrid
PolarAxesGrid
AntennaGrid

Axis grids are solid, dotted or dashed lines, aligned with the axis tick marks and which extend across the plot area of a graph. Axis grids are a separate class from the axis classes. An axis class, i.e. any class derived from **Axis**, can exist independent of a grid. Many graphs use axes that do not have grids. For example, the y-axis may have a grid, while the x-axis may not. The axis grid classes are not independent and they require valid references to both an x- and y-axis. The three axis grid classes are **ChartGrid, PolarGrid** and **AntennaGrid.**

Linear, Logarithmic and Time Axis Grids

Class ChartGrid

GraphObj
|
+-- ChartGrid

The **ChartGrid** class defines a grid for **LinearAxis**, **LogAxis**, and **TimeAxis** classes. A grid object needs a reference to both an x- and y-axis. The grid aligns with the tick marks of one axis, and extends across the plot area using the minimum and maximum values of the other axis. For example, an x-axis grid has lines aligned with the tick marks of the x-axis and extending from the minimum y-value to the maximum y-value of the y-axis, parallel to the y-axis.

ChartGrid constuctor

```
[C#]
public ChartGrid(
   Axis xaxis,
   Axis yaxis,
   int gridaxistype,
   int gridtype
);
```

xaxis The x-axis associated with the grid.

yaxis	The y-axis associated with the grid.
gridaxistype	The grid is aligned with the tick marks of this axis. The grid is parallel to the other axis. Use one of the axis constants, X_AXIS or Y_AXIS.
gridtype	Specifies if the grid aligns with the major tick marks (GRID_MAJOR), the minor tick marks (GRID_MINOR) or the major and minor tick marks (GRID_ALL) of the reference axis.

Other grid properties are associated with the line properties used to draw the grid. The default values of the grid use a black dotted line of thickness 1.0. Change the default values using the **GraphObj** methods below.

SetColor method

```
[C#]
public virtual void SetColor(
   Color rgbcolor
);
```

SetLineWidth method

```
[C#]
public virtual void SetLineWidth(
   double linewidth
);
```

SetLineStyle method

```
[C#]
public virtual void SetLineStyle(
   DashStyle linestyle
);
```

rgbcolor	Sets the primary line color for the chart object.
linewidth	Sets the line width, in device coordinates, for the chart object.
linestyle	Sets the line style for the chart object. Use one of the ChartObj line style enumerated constants: LS_SOLID, LS_DASH_4_4, LS_DASH_4_2, LS_DASH_2_2, LS_DOT_1_1, LS_DOT_1_2, LS_DOT_1_4, LS_DOT_1_8, LS_DASH_DOT.

ChartGrid example

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
// Define the coordinate system
double xMin = -5;
double xMax = 15;
double yMin = 0;
double yMax =  105;
CartesianCoordinates simpleScale =
       new CartesianCoordinates(xMin, yMin, xMax, yMax);

// Create the x- and y-axes
LinearAxis xAxis =
new LinearAxis(simpleScale, ChartObj.X_AXIS);
LinearAxis yAxis =
new LinearAxis(simpleScale, ChartObj.Y_AXIS);
// Create the grids
ChartGrid gridX =  new ChartGrid(xAxis, yAxis, ChartObj.X_AXIS, ChartObj.GRID_MAJOR);
//Change default grid line properties
gridX.SetLineWidth(2.0);
gridX.SetLineStyle(ChartObj.LS_DOT_1_4);
gridX.SetColor(Colors.Gray);

ChartGrid gridY =  new ChartGrid(xAxis, yAxis, ChartObj.Y_AXIS, ChartObj.GRID_MAJOR);
//Change default grid line properties
gridY.SetLineWidth(1.0);
gridY.SetLineStyle(ChartObj.LS_SOLID);
gridY.SetColor(Colors.Black);


// Add the x- and y-axes to the chartVu object
chartVu.AddChartObject(xAxis);
chartVu.AddChartObject(yAxis);
chartVu.AddChartObject(gridX);
chartVu.AddChartObject(gridY);
```

Polar Grids

Class PolarGrid

GraphObj
|
+-- ChartGrid
|
+-- PolarGrid

The **PolarGrid** class defines a grid for polar axes. The polar grid consists of two parts: the magnitude grid and the polar angle grid. The magnitude grid consists of a group of concentric

circles centered on the origin and aligned with the x- and y-axis tick marks. The polar angle grid consists of a group of radial lines, aligned with the angle tick marks and starting at the origin extending to the outer polar angle circle.

PolarGrid constructors

There are two **PolarGrid** constructors.

```
[C#]
public PolarGrid(
   PolarAxes polaraxis,
   int gridtype
);

public PolarGrid(
   PolarAxes polaraxis,
   int gridmagtype,
   int gridangletype
);
```

polaraxis	The polar axes associated with the grid.
gridtype	Specifies if the magnitude and angular grid aligns with the major tick marks (GRID_MAJOR), the minor tick marks (GRID_MINOR) or the major and minor tick marks (GRID_ALL) of the reference polar axis.
gridmagtype	Specifies if the magnitude grid aligns with the major tick marks (GRID_MAJOR), the minor tick marks (GRID_MINOR) or the major and minor tick marks (GRID_ALL) of the reference polar axis.
gridangletype	Specifies if the angular grid aligns with the major tick marks (GRID_MAJOR), the minor tick marks (GRID_MINOR) or the major and minor tick marks (GRID_ALL) of the reference polar axis.

The **SetLineWidth**, **SetLineStyle** and **SetColor** methods are used to customize the drawing properties of the lines used to draw the axes lines and tick marks.

Polar grid example

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
double polarmagnitude = 5.0;
PolarCoordinates polarscale = new PolarCoordinates(polarmagnitude);
```

```
PolarAxes polarAxes = new PolarAxes(polarscale);
PolarGrid polarGrid =  new PolarGrid(polarAxes, ChartObj.GRID_MAJOR);
// Create the ChartView object to place graph objects in.
ChartView chartVu  = new ChartView();
// Add the polar axes to the chartVu object
chartVu.AddChartObject(polarAxes);
chartVu.AddChartObject(polarGrid);
```

Antenna Grids

Class AntennaGrid

GraphObj
|
+-- ChartGrid
|
+-- AntennaGrid

The **AntennaGrid** class defines a grid for antenna axes. The antenna grid consists of two parts: the circular grid and the radial grid. The circular grid consists of a group of concentric circles centered on the origin and aligned with the y-axis tick marks. The antenna radial grid consists of a group of radial lines, aligned with the angle tick marks and starting at the origin extending to the outer edge of the antenna coordinate system.

AntennaGrid constructors

There are two **AntennaGrid** constructors.

```
[C#]
public AntennaGrid(
   AntennaAxes baseaxis,
   int gridtype
);

public AntennaGrid (
   AntennaAxes baseaxis,
   int gridmagtype,
   int gridangletype
);
```

baseaxis The antenna axes associated with the grid.

gridtype	Specifies if the radial and angular grid aligns with the major tick marks (GRID_MAJOR), the minor tick marks (GRID_MINOR) or the major and minor tick marks (GRID_ALL) of the reference antenna axis.
gridmagtype	Specifies if the radial grid aligns with the major tick marks (GRID_MAJOR), the minor tick marks (GRID_MINOR) or the major and minor tick marks (GRID_ALL) of the reference antenna axis.
gridangletype	Specifies if the angular grid aligns with the major tick marks (GRID_MAJOR), the minor tick marks (GRID_MINOR) or the major and minor tick marks (GRID_ALL) of the reference antenna axis.

The **SetLineWidth**, **SetLineStyle** and **SetColor** methods are used to customize the drawing properties of the lines used to draw the axes lines and tick marks.

Antenna grid example

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
AntennaAxes pAntennaAxis = pAntennaTransform.GetCompatibleAxes();
pAntennaAxis.LineColor = Colors.Black;
chartVu.AddChartObject(pAntennaAxis);

AntennaGrid pAntennaGrid = new AntennaGrid(pAntennaAxis, AntennaGrid.GRID_ALL);
pAntennaGrid.ChartObjAttributes = new ChartAttribute(Colors.LightBlue, 1,
ChartObj.LS_SOLID);
chartVu.AddChartObject(pAntennaGrid);
```

10. Simple Plot Objects

SimplePlot
- **SimpleBarPlot**
- **SimpleLineMarkerPlot**
- **SimpleLinePlot**
- **SimpleScatterPlot**
- **SimpleVersaPlot**

The **SimplePlot** class is an abstract class representing plot types that use data organized as a simple array of xy points, where there is one y for every x. Simple plot types include: line plots, scatter plots, bar graphs, and line-marker plots. When used in the simplest mode, simple plot objects use a single **ChartAttribute** object to control the plot objects color, line, and gradient styles. In terms of memory usage, this is the most efficient method. If memory is not an issue, it is also possible to assign every line segment, bar and scatter plot symbol a unique **ChartAttribute** object. Used in this mode, a single line plot can have unlimited number of multi-colored line segments. Another option labels each data point with its numeric y-value.

Example program segments presented in this documentation are not complete programs and contain uninitialized and/or undefined objects and variables. Do not attempt to copy them into your own program. Refer to the referenced examples.

Simple Line Plots

Class SimpleLinePlot

GraphObj
- **ChartPlot**
  - **SimplePlot**
    - **SimpleLinePlot**

The **SimpleLinePlot** class is a concrete implementation of the **SimplePlot** class and and displays simple datasets in line plot format. Data points are connected using a straight line, or a step line.

SimpleLinePlot constructor

```
[C#]
public SimpleLinePlot(
   PhysicalCoordinates transform,
   SimpleDataset dataset,
```

```
   ChartAttribute attrib
);
```

transform	The coordinate system for the new **SimpleLinePlot** object.
dataset	The line plot represents the values in this dataset.
attrib	Specifies the attributes (line color, thickness and style, fill color and fill mode) for the line plot.

A **ChartAttribute** object sets the objects global line color, line thickness, line style, fill color and fill mode. Change the **ChartAttribute** object using the objects **SetChartObjAttributes** method. There is also a group of methods that set individual simple plot properties: **SetColor**, **SetLineWidth**, and **SetLineStyle**. The line step style is using the **SetStepMode** method.

Individual line segments in a simple line plot object can have unique properties. Use the objects **SetSegmentAttributesMode** and **SetSegmentAttributes** methods.

Simple line plot example (extracted from the example program SimpleLinePlots, class LineFill)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
TimeCoordinates  pTransform1 = new TimeCoordinates();
pTransform1.AutoScale(DatasetArray,
                    ChartObj.AUTOAXES_FAR, ChartObj.AUTOAXES_FAR);
.
.
.

ChartAttribute attrib1 =
           new ChartAttribute (Colors.Blue, 3,ChartObj.ChartObj.LS_SOLID);
SimpleLinePlot thePlot1 =
           new SimpleLinePlot(pTransform1, Dataset1, attrib1);
thePlot1.SetLineStyle(DashDot);
chartVu.AddChartObject(thePlot1);
```

Simple line plot example using segment colors (extracted from the example program SimpleLinePlots, class LineFill)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
           TimeSimpleDataset[] DatasetArray = { Dataset1, Dataset2, Dataset3 };

            pTransform1 = new TimeCoordinates();
            pTransform1.AutoScale(DatasetArray, ChartObj.AUTOAXES_FAR,
ChartObj.AUTOAXES_FAR);

            pTransform1.SetGraphBorderDiagonal(0.15, .1, .92, 0.75);
           Background background = new Background(pTransform1, ChartObj.GRAPH_BACKGROUND,
                        Colors.Coral, Colors.Blue, ChartObj.Y_AXIS);
            //         Background background = new Background(pTransform1,
ChartObj.GRAPH_BACKGROUND,
            //              ChartColor.FromRgb(100, 50, 255), ChartColor.FromRgb(40, 25,
120), ChartObj.Y_AXIS);
          //   Background background = new Background(pTransform1,
ChartObj.GRAPH_BACKGROUND, Colors.Black);

            chartVu.AddChartObject(background);

            Background plotbackground = new Background(pTransform1,
ChartObj.PLOT_BACKGROUND,
                ChartColor.Black);
            chartVu.AddChartObject(plotbackground);

            TimeAxis xAxis = new TimeAxis(pTransform1);
            xAxis.SetColor(ChartColor.White);
            chartVu.AddChartObject(xAxis);

            LinearAxis yAxis = new LinearAxis(pTransform1, ChartObj.Y_AXIS);
            yAxis.SetColor(ChartColor.White);
            chartVu.AddChartObject(yAxis);

            LinearAxis yAxis2 = new LinearAxis(pTransform1, ChartObj.Y_AXIS);
            yAxis2.SetAxisIntercept(xAxis.GetAxisMax());
            yAxis2.SetAxisTickDir(ChartObj.AXIS_MAX);
            yAxis2.SetColor(ChartColor.White);
            chartVu.AddChartObject(yAxis2);

            TimeAxisLabels xAxisLab = new TimeAxisLabels(xAxis);
            xAxisLab.SetAxisLabelsFormat(ChartObj.TIMEDATEFORMAT_Y2000);
            xAxisLab.SetColor(ChartColor.White);
            chartVu.AddChartObject(xAxisLab);

            NumericAxisLabels yAxisLab = new NumericAxisLabels(yAxis);
            yAxisLab.SetColor(ChartColor.White);
            yAxisLab.SetAxisLabelsFormat(ChartObj.CURRENCYFORMAT);
            chartVu.AddChartObject(yAxisLab);

            AxisTitle yaxistitle = new AxisTitle(yAxis, theFont, "Millions $");
            yaxistitle.SetColor(ChartColor.White);
            chartVu.AddChartObject(yaxistitle);

            ChartGrid xgrid = new ChartGrid(xAxis, yAxis, ChartObj.X_AXIS,
ChartObj.GRID_MAJOR);
            xgrid.SetColor(ChartColor.White);
            chartVu.AddChartObject(xgrid);

            ChartGrid ygrid = new ChartGrid(xAxis, yAxis, ChartObj.Y_AXIS,
ChartObj.GRID_MAJOR);
            ygrid.SetColor(ChartColor.White);
            chartVu.AddChartObject(ygrid);

            ChartAttribute attrib1 = new ChartAttribute(ChartColor.Blue, 3,
ChartObj.LS_SOLID);
```

```
            Dataset1.SortByX(true);
            thePlot1 = new SimpleLinePlot(pTransform1, Dataset1, attrib1);
            thePlot1.SetLineStyle(ChartObj.LS_DASH_DOT);
            chartVu.AddChartObject(thePlot1);

            ChartAttribute attrib2 = new ChartAttribute(ChartColor.Yellow, 3,
ChartObj.LS_SOLID);
            Dataset2.SortByX(true);
            thePlot2 = new SimpleLinePlot(pTransform1, Dataset2, attrib2);
            chartVu.AddChartObject(thePlot2);

            Color transparentRed = Color.FromArgb(200, 255, 0, 0);
            Color transparentGreen = Color.FromArgb(200, 0, 255, 0);

            ChartAttribute lossAttrib = new ChartAttribute(transparentRed, 1,
ChartObj.LS_SOLID,
                transparentRed);
            ChartAttribute profitAttrib = new ChartAttribute(transparentGreen, 1,
ChartObj.LS_SOLID,
                transparentGreen);

            profitAttrib.SetFillFlag(true);
            lossAttrib.SetFillFlag(true);
            profitAttrib.SetLineFlag(false);
            lossAttrib.SetLineFlag(false);
            // Must call the linePlot (or similar function) before setting segment
            // attributes so that it know size of segment buffer to allocate.
            thePlot3 = new SimpleLinePlot(pTransform1, Dataset3, profitAttrib);
            double[] yValues = Dataset3.GetYData();
            thePlot3.SetSegmentAttributesMode(true);
            for (i = 0; i < Dataset3.GetNumberDatapoints(); i++)
            {
                if (yValues[i] > 0.0)
                    thePlot3.SetSegmentAttributes(i, profitAttrib);
                else
                    thePlot3.SetSegmentAttributes(i, lossAttrib);
            }
            chartVu.AddChartObject(thePlot3);
```

Simple Bar Plots

Class SimpleBarPlot

GraphObj

|

+--ChartPlot

|

+--SimplePlot

|

+--SimpleBarPlot

The **SimpleBarPlot** class is a concrete implementation of the **SimplePlot** class and displays data in a bar format. Individual bars, the maximum value of which corresponds to the y-values of the dataset, display justified with respect to the x-values.

SimpleBarPlot constructor

```
[C#]
public SimpleBarPlot(
   PhysicalCoordinates transform,
   SimpleDataset dataset,
   double barwidth,
   double barbase,
   ChartAttribute attrib,
   int barjust
);
```

transform	The coordinate system for the new **SimpleBarPlot** object.
dataset	The bar plot represents the values in this dataset.
barwidth	The width of the bars in physical coordinates.
barbase	The base value for bars in physical coordinates.
attrib	Specifies the attributes (line color and fill color) of the bars.
barjust	Specifies the justification with respect to the independent data value. Use one of the justification constants: JUSTIFY_MIN, JUSTIFY_CENTER, JUSTIFY_MAX.

A **ChartAttribute** object sets the objects global line color, line thickness, line style, fill color and fill mode. Change the **ChartAttribute** object using the objects **SetChartObjAttributes** method. The simple bar plot **SetColor** method can be used to change the bar color.

Individual bars in a simple bar plot object can have unique properties. Use the objects **SetSegmentAttributesMode** and **SetSegmentAttributes** methods.

Simple bar plot example (extracted from the example program Bargraphs, class SimpleBars)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
TimeSimpleDataset Dataset1 = new TimeSimpleDataset("Actual Sales",x1,y1);
TimeCoordinates pTransform1 = new TimeCoordinates();
pTransform1.AutoScale(Dataset1, ChartObj.AUTOAXES_FAR , ChartObj.AUTOAXES_FAR);
pTransform1.SetScaleStartY(0);
pTransform1.SetTimeScaleStart(new ChartCalendar(1997,ChartObj.JULY,1));
pTransform1.SetGraphBorderDiagonal(0.15, .15, .9, 0.8) ;
Background background = new Background( pTransform1, ChartObj.GRAPH_BACKGROUND,
           Color.FromArgb(255,30,70,70), Color.FromArgb(255,90,20,155), ChartObj.Y_AXIS);
chartVu.AddChartObject(background);
.
.  // Define and add axes, axes labels and grids to chart
```

```
.
ChartAttribute attrib1 =
        new ChartAttribute (Colors.Green, 0,ChartObj.LS_SOLID, Colors.Green);
attrib1.SetFillFlag(true);
SimpleBarPlot thePlot1 = new SimpleBarPlot(pTransform1, Dataset1,
        ChartCalendar.GetCalendarWidthValue(ChartObj.MONTH,8), 0.0,
        attrib1, ChartObj.JUSTIFY_CENTER);
chartVu.AddChartObject(thePlot1);
```

* Note how the **ChartCalendar.GetCalendarWidthValue** method calculates the width of the bars as a function of time, in this case a width of 8 months.

Simple bar plot example that displays numeric data values (extracted from the example program Bargraphs, class SimpleBars)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
ChartAttribute attrib1 =
        new ChartAttribute (Colors.Green, 0,ChartObj.LS_SOLID, Colors.Green);
attrib1.SetFillFlag(true);
SimpleBarPlot thePlot1 = new SimpleBarPlot(pTransform1, Dataset1,
        ChartCalendar.GetCalendarWidthValue(ChartObj.MONTH,8), 0.0,
        attrib1, ChartObj.JUSTIFY_CENTER);
NumericLabel bardatavalue = thePlot1.GetPlotLabelTemplate();
bardatavalue.SetTextFont(theFont);
bardatavalue.SetNumericFormat(ChartObj.CURRENCYFORMAT);
bardatavalue.SetDecimalPos(0);
bardatavalue.SetColor(Colors.White);
thePlot1.SetPlotLabelTemplate(bardatavalue);
thePlot1.SetShowDatapointValue(true);
chartVu.AddChartObject(thePlot1);
```

Simple Scatter Plots

Class SimpleScatterPlot

GraphObj
|
+--ChartPlot
|
+--SimplePlot
|
+--SimpleScatterPlot

The **SimpleScatterPlot** class is a concrete implementation of the **SimplePlot** class and displays simple datasets in scatter plot format where each data point is a symbol.

SimpleScatterPlot constructor

```
[C#]
public SimpleScatterPlot(
   PhysicalCoordinates transform,
   SimpleDataset dataset,
   int symtype,
   ChartAttribute attrib
);
```

transform	The coordinate system for the new **SimpleScatterPlot** object.
dataset	The scatter plot represents the values in this dataset.
symtype	The symbol used in the scatter plot. Use one of the scatter plot symbol constants: NOSYMBOL, SQUARE, TRIANGLE, DIAMOND, CROSS, PLUS, STAR, LINE, HBAR, VBAR, CIRCLE.
attrib	Specifies the attributes (size, line and fill color) for the scatter plot.

A **ChartAttribute** object sets the objects global outline and fill attributes. Change the simple plot objects **ChartAttribute** object using the objects **SetChartObjAttributes** method. For a simple color change of the scatter plot symbol, use the scatter plot objects **SetColor** method.

Should you need additional symbols, create your own. Any **PathGeometry** object can be used as a symbol. The coordinates of the symbol should assume that 1.0 is the standard symbol size with a symbol center at the relative coordinates (0.5, 0.5). The example below demonstrates how to create a diamond symbol.

```
public PathGeometry GetDiamondShape()
{
  PathGeometry result = ChartSupport.NewPath();

  ChartSupport.StartPath(result, new Point(0.5, 0.0), false);
  ChartSupport.AddPointToPath(result, new Point(0.0, 0.5));
  ChartSupport.AddPointToPath(result, new Point(0.5, 1.0));
  ChartSupport.AddPointToPath(result, new Point(1.0, 0.5));
  ChartSupport.AddPointToPath(result, new Point(0.5, 0.0));
  ChartSupport.CloseCurrentFigure(result);

   result.Figures[0].IsClosed = true;

  return (PathGeometry)result;
}
```

Set the custom symbol using **SimpleScatterPlot.SetCustomScatterPlotSymbol** method after the **SimpleScatterPlot** object is created.

Individual scatter plot symbols in a scatter plot object can have unique properties. Use the objects **SetSegmentAttributesMode** and **SetSegmentAttributes** methods.

Simple scatter plot example (extracted from the example program ScatterPlots, class SimpleScatter)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
pTransform1.SetGraphBorderDiagonal(0.15, .15, .90, 0.725) ;
Background background =
    new Background( pTransform1, ChartObj.PLOT_BACKGROUND, Colors.White);
chartVu.AddChartObject(background);
.
. // Define and add axes, axes labels and grids to chart
.
ChartAttribute attrib1 = new ChartAttribute (Colors.Blue, 1,ChartObj.LS_SOLID);
attrib1.SetFillColor (Colors.Blue);
attrib1.SetFillFlag (true);
attrib1.SetSymbolSize(10);
SimpleScatterPlot thePlot1 =
        new SimpleScatterPlot(pTransform1, Dataset2,  ChartObj.CROSS, attrib1);
chartVu.AddChartObject(thePlot1);

ChartAttribute attrib3 = new ChartAttribute (Colors.Red, 1,ChartObj.LS_SOLID);
attrib3.SetFillColor (Colors.Red);
attrib3.SetFillFlag (true);
attrib3.SetSymbolSize(6);
SimpleScatterPlot thePlot3 =
     new SimpleScatterPlot(pTransform1, Dataset3,  ChartObj.CIRCLE, attrib3);
chartVu.AddChartObject(thePlot3);
```

Simple scatter plot example that uses SetSegmentAttributesMode to change the size and color of individual scatter plot symbols in the plot (extracted from the example program ScatterPlots, class ScatterPoints)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
ChartAttribute attrib1 = new ChartAttribute (Colors.Blue, 1,ChartObj.LS_SOLID);
attrib1.SetFillColor (Colors.Blue);
attrib1.SetLineFlag(true);
attrib1.SetSymbolSize(10);
SimpleScatterPlot thePlot1 =
    new SimpleScatterPlot(pTransform1, Dataset1,  ChartObj.SQUARE, attrib1);
thePlot1.SetSegmentAttributesMode(true);
ChartAttribute segmentAttrib =
    new ChartAttribute (Colors.Red, 1,ChartObj.LS_SOLID, Colors.Red);
segmentAttrib.SetSymbolSize(20);
thePlot1.SetSegmentAttributes(8,segmentAttrib);
thePlot1.SetSegmentAttributes(9,segmentAttrib);
thePlot1.SetSegmentAttributes(10,segmentAttrib);
thePlot1.SetSegmentAttributes(11,segmentAttrib);
chartVu.AddChartObject(thePlot1);
```

Simple Line Marker Plots

Class SimpleLineMarkerPlot

GraphObj
|
+--ChartPlot
|
+--SimplePlot
|
+-- SimpleLineMarkerPlot

The **SimpleLineMarkerPlot** class is a concrete implementation of the **SimplePlot** class and displays simple datasets in a line plot format where scatter plot symbols highlight individual data points.

SimpleLineMarkerPlot constructor

```
[C#]
public SimpleLineMarkerPlot(
   PhysicalCoordinates transform,
   SimpleDataset dataset,
   int symtype,
   ChartAttribute lineattrib,
   ChartAttribute symbolattrib,
   int nsymbolstart,
   int nsymbolskip
);
```

transform	The coordinate system for the new **SimpleLineMarkerPlot** object.
dataset	The line marker plot represents the values in this dataset.
symtype	The symbol used in the line marker plot. Use one of the scatter plot symbol constants: NOSYMBOL, SQUARE, TRIANGLE, DIAMOND, CROSS, PLUS, STAR, LINE, HBAR, VBAR, CIRCLE.
lineattrib	Specifies the attributes (line color and line style) for the line part of the line marker plot.
symbolattrib	Specifies the attributes (line and fill color) for the symbol part of the line marker plot.
nsymbolstart	Specifies the starting index for symbols in the line marker plot.
nsymbolskip	Specifies the skip factor for placing symbols in the line marker plot.

An **ChartAttribute** object sets the objects global line color, line width and line style attributes. Change the **ChartAttribute** object using the objects **SetChartObjAttributes** method. Use the objects **setSymbolAttributes** to change the attributes of the marker symbol.

Should you need additional symbols, create your own. Any **PathGeometry** object can be used as a symbol. The coordinates of the symbol should assume that 1.0 is the standard symbol size with a symbol center at the relative coordinates (0.5, 0.5). See the example in the discussion of the **SimpleScatterPlot** class.

Individual line segments in a line marker plot object can have unique properties. Use the objects **SetSegmentAttributesMode** and **SetSegmentAttributes** methods. If this option is used, the line and fill properties of the lines and the marker symbols will be the same.

Simple line marker plot example (extracted from the example program ScatterPlots, class LabeledDatapoints)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
SimpleDataset Dataset1 = new SimpleDataset("First",x1,y1);
CartesianCoordinates pTransform1 =
       new CartesianCoordinates( ChartObj.LINEAR_SCALE, ChartObj.LINEAR_SCALE);
pTransform1.AutoScale(Dataset1, ChartObj.AUTOAXES_FAR, ChartObj.AUTOAXES_FAR);

pTransform1.SetGraphBorderDiagonal(0.15, .15, .90, 0.7) ;
.
. // Define and add axes, axes labels and grids to chart
.
ChartAttribute attrib1 = new ChartAttribute (Colors.Blue, 1,ChartObj.LS_SOLID);
ChartAttribute attrib2 = new ChartAttribute (Colors.Blue, 1,ChartObj.LS_SOLID);
attrib2.SetFillColor (Colors.Blue);
attrib2.SetFillFlag (true);
attrib2.SetSymbolSize(10);
SimpleLineMarkerPlot thePlot1 =
               new SimpleLineMarkerPlot(pTransform1, Dataset1,
               ChartObj.SQUARE, attrib1,attrib2, 0, 1);
chartVu.AddChartObject(thePlot1);
```

Add the following lines to the program segment above to add data point labeling to the line marker plot.

[C#]

```
thePlot1.SetShowDatapointValue(true);
NumericLabel modellabel = new NumericLabel();
modellabel.SetXJust(ChartObj.JUSTIFY_CENTER);
```

```
modellabel.SetYJust(ChartObj.JUSTIFY_MIN);
ChartFont modellabelfont = new ChartFont("Microsoft Sans Serif", 10, FontStyle.Normal);
modellabel.SetTextFont(modellabelfont);
modellabel.SetTextNudge(0,-5);
thePlot1.SetPlotLabelTemplate(modellabel);
```

Simple Versa Plots

Class SimpleVersaPlot

GraphObj

|

+--ChartPlot

|

+--SimplePlot

|

+-- SimpleLineMarkerPlot

|

+-- SimpleVersaPlot

The **SimpleVersaPlot** is a plottype that can be any of the four simple plot types: LINE_MARKER_PLOT, LINE_PLOT, BAR_PLOT, SCATTER_PLOT. It is used when you want to be able to change from one plot type to another, without deleting the instance of the old plot object and creating an instance of the new.

SimpleLineMarkerPlot constructor

```
[C#]

public SimpleVersaPlot(
   PhysicalCoordinates transform,
   SimpleDataset dataset,
   ChartAttribute lineattrib
);

public SimpleVersaPlot(
   PhysicalCoordinates transform,
   SimpleDataset dataset,
   Double barwidth,
   Double barbase,
   ChartAttribute attrib,
   Int32 barjust
);

public SimpleVersaPlot(
   PhysicalCoordinates transform,
   SimpleDataset dataset,
   Int32 symtype,
```

```
   ChartAttribute symbolattrib
);
public SimpleVersaPlot(
   PhysicalCoordinates transform,
   SimpleDataset dataset,
   Int32 symtype,
   ChartAttribute lineattrib,
   ChartAttribute symbolattrib,
   Double barwidth
);
```

transform	The coordinate system for the new **SimpleVersaPlot** object.
dataset	The versa plot represents the values in this dataset.
symtype	The symbol used in the line marker plot. Use one of the scatter plot symbol constants: NOSYMBOL, SQUARE, TRIANGLE, DIAMOND, CROSS, PLUS, STAR, LINE, HBAR, VBAR, CIRCLE.
lineattrib	Specifies the attributes (line color and line style) for the line part of the line marker plot.
symbolattrib	Specifies the attributes (line and fill color) for the symbol part of the line marker plot.
nsymbolstart	Specifies the starting index for symbols in the line marker plot.
nsymbolskip	Specifies the skip factor for placing symbols in the line marker plot.
barwidth	Specifies the width of the bar.

A **ChartAttribute** object sets the objects global line color, line width and line style attributes. Change the **ChartAttribute** object using the objects **SetChartObjAttributes** method. Use the objects **setSymbolAttributes** to change the attributes of the marker symbol.

Change the plot type using the PlotType property. Use one of the plot type constants: LINE_MARKER_PLOT, LINE_PLOT, BAR_PLOT, SCATTER_PLOT.

Simple versa-plot example (extracted from the example program NewDemosRev2.SimpleVersaChart)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
ChartAttribute attrib1 = new ChartAttribute(Colors.Green, 0, ChartObj.LS_SOLID,
Colors.Green);
Color[] barcolors = { Colors.Red, Colors.Orange, Colors.Yellow, Colors.White };
```

```
double[] barbreakpoints = { 0.0, 80, 160, 240 };
int gradmode = ChartGradient.GRADIENT_MAPTO_PLOT_PHYSICAL_COORDINATES;
ChartGradient cg = new ChartGradient(pTransform1, gradmode,
    barcolors, barbreakpoints, -90);
attrib1.Gradient = cg;

attrib1.SetFillFlag(true);
thePlot1 = new SimpleVersaPlot(pTransform1, Dataset1,
   ChartCalendar.GetCalendarWidthValue(ChartObj.MONTH, 8), 0.0,
   attrib1, ChartObj.JUSTIFY_CENTER);
NumericLabel bardatavalue = thePlot1.GetPlotLabelTemplate();
bardatavalue.SetTextFont(theFont);
bardatavalue.SetNumericFormat(ChartObj.CURRENCYFORMAT);
bardatavalue.SetDecimalPos(0);
bardatavalue.SetColor(Colors.White);
thePlot1.SetPlotLabelTemplate(bardatavalue);
thePlot1.SetShowDatapointValue(true);
chartVu.AddChartObject(thePlot1);
```

11. Group Plot Objects

GroupPlot

- **ArrowPlot**
- **BubblePlot**
- **CandlestickPlot**
- **CellPlot**
- **ErrorBarPlot**
- **FloatingBarPlot**
- **GroupBarPlotChartPlot**
- **HistogramPlot**
- **LineGapPlot**
- **MultiLinePlot**
- **OHLCPlot**
- **StackedBarPlot**
- **StackedLinePlot**

The **GroupPlot** class is an abstract class representing plot types that use data organized as arrays of x- and y-values, where there is one or more y-value for each x-value. Group plot types include: multi-line plots, stacked line plots, stacked bar plots, group bar plots, error bar plots, floating bar plots, open-high-low-close plots, candlestick plots, arrow plots, histogram plots, cell plots and bubble plots.

The number of x-values in a group plot is referred to as the number of columns, or as **NumberDatapoints** and the number of y-values *for each x-value* is referred to as the number of rows, or **NumberGroups**. Think of spreadsheet that looks like:

x-values	x[0]	x[1]	x[2]	x[3]	x[4]	x[5]
y-values group #0	y[0,0]	y[0,1]	y[0,2]	y[0,3]	y[0,4]	y[0,5]
y-values group #1	y[1,0]	y[1,1]	y[1,2]	y[1,3]	y[1,4]	y[1,5]
y-values group #2	y[2,0]	y[2,1]	y[2,2]	y[2,3]	y[2,4]	y[2,5]

number of x-values = **NumberDatapoints** = NumberColumns = 6

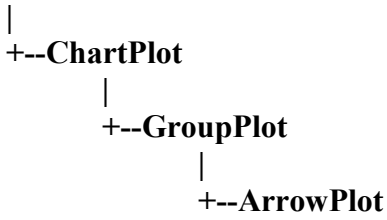
number of y-values for each x-value = **NumberGroups** = NumberRows = 3

This would be the ROW_MAJOR format if the data were stored in a CSV file.

Example program segments presented in this documentation are not complete programs and contain uninitialized and/or undefined objects and variables. Do not attempt to copy them into your own program. Refer to the referenced example program that the code is extracted from.

Arrow Plots

Class ArrowPlot

GraphObj


The **ArrowPlot** class is a concrete implementation of the **GroupPlot** class. It displays a collection of arrows as defined by the data in a group dataset. The position, size, and rotation of each arrow in the collection is independently controlled. . The numer of groups of the group dataset must be three.

ArrowPlot constructor

```
[C#]
public ArrowPlot(
   PhysicalCoordinates transform,
   GroupDataset dataset,
   Arrow basearrow,
   ChartAttribute attrib
);
```

transform The coordinate system for the new **ArrowPlot** object.

dataset The group dataset sets the position, size and rotation of individual arrows. The numer of groups must be three. Orgainize the data in the dataset in the following manner:

X x-position of the arrow point.

Y[0] y-position of the arrow point.

Y[1] Size of the arrow. A size of 0.05 creates an arrow with a length equal to 0.05 in NORM_PLOT_POS coordinates.

	Y[2] The rotation of the arrow, using the point of the arrow as the rotation origin, in degrees.
basearrow	An instance of an **Arrow** object used to draw the arrows in this **ArrowPlot** object.
attrib	Sets the color, line and fill characteristics for the arrows in this **ArrowPlot** object.

An individual arrow in an arrow plot object can have unique attributes. Use the objects **SetSegmentAttributesMode** and **SetSegmentAttributes** methods in the manner described for **SimplePlot** objects..

Arrow plot example (extracted from the example program ScatterPlots, class ArrowChart)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
GroupDataset Dataset1 = new GroupDataset("First",x1,y1);
Dataset1.SetAutoScaleNumberGroups(1);
CartesianCoordinates pTransform1 =
new CartesianCoordinates( ChartObj.LINEAR_SCALE, ChartObj.LINEAR_SCALE);
pTransform1.AutoScale(Dataset1, ChartObj.AUTOAXES_FAR, ChartObj.AUTOAXES_FAR);
pTransform1.SetScaleX(0,10);
pTransform1.SetScaleY(0,10);
pTransform1.SetGraphBorderDiagonal(0.15, .15, .90, 0.75) ;
.
.       // Define axes, axes labels and grids
.
ChartAttribute attrib1 = new ChartAttribute (Colors.Blue, 1,ChartObj.LS_SOLID);
attrib1.SetFillColor (Colors.Blue);
attrib1.SetFillFlag (true);
Arrow basearrow = new Arrow();
ArrowPlot thePlot1 = new ArrowPlot(pTransform1, Dataset1, basearrow, attrib1);
chartVu.AddChartObject(thePlot1);
```

*Note the use of the **GroupDataset** method **SetAutoScaleNumberGroups**. This forces the auto-scale routine to look at only the first group of y-values, since those are the only y-values that specify the absolute position of the arrows. The other groups of y-values specify size and rotation information and should not be considered in the auto-scale calculation.

Box and Whisker Plots

Class BoxWhiskerPlot

GraphObj

```
|
+--ChartPlot
        |
        +--GroupPlot
                |
                +-- BoxWhiskerPlot
```

The **BoxWhiskerPlot** class extends the **GroupPlot** class and displays statistical data in a box and whisker format. The **BoxWhiskerPlot** class graphically represents groups of numerical data through their five-number summaries (the smallest observation, lower quartile (Q1), median (Q2), upper quartile (Q3), and largest observation). A **BoxWhiskerPlot** is unique among our chart types because the data is not represented by a 1D or 2D matrix. Instead, it consists of multiple populations, where each population can have a different number of data points. Each population is summarized by 5 statistics (the smallest observation, lower quartile (Q1), median (Q2), upper quartile (Q3), and largest observation), display graphically in a chart. Read the Wikipedia entry for more information concerning box and whisker plots: http://en.wikipedia.org/wiki/Box_plot

BoxWhiskerPlot constructor

```
C#
public BoxWhiskerPlot(
    PhysicalCoordinates transform,
    double rwidth,
    ChartAttribute attrib
)
```

transform The coordinate system for the new **BoxWhiskerPlot** object.

rwidth The width of the candlestick box in physical coordinates.

attrib Specifies the attributes (line color and fill color) of the candlestick lines when the close value is greater than the open value.

Once the initial **BoxWhiskerPlot** object is created, populations are added to it, one population at a time, using the **AddPopulation** method. Each population can have a different number of data points. Once all of the population groups are added, the software will calculate the quartile data for each population in response to the **AutoBWChart** method call. Each population is summarized by a single box in the box and whisker plot.

```
C#
public void AddPopulation(
    double[] pop,
    double xvalue
)
```

Parameters

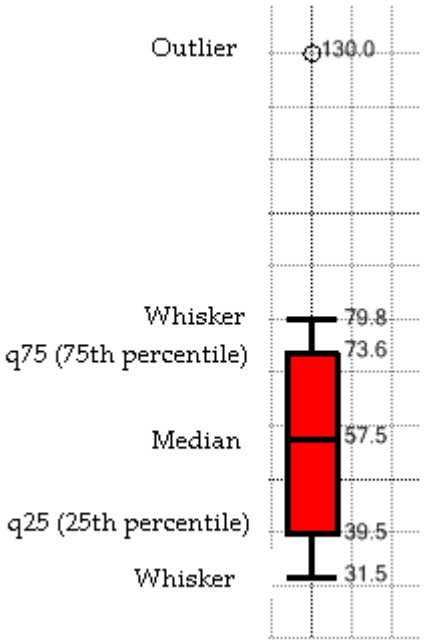
pop	The source population of y-values to add.
xvalue	The x-value of the of y-values population.

There are several variants of box and whisker plots. Select which one you want using the **BWFormat** property. Use one of the BW format constants: BW_MINMAX_WHISKER, BW_IRQ15_WHISKER_OUTLIERS, BW_IQR15_WHISKER_ALLPOINTS.

BW_MINMAX_WHISKER	Plot minimum, maximum values as whiskers, and the median, q25 and q75 values as the box.
BW_IQR15_WHISKER_OUTLIERS	Plot the minimum value within q25 - 1.5*IQR and maximum value within q75 + 1.5*IQR as whiskers, the median, q25 and q75 values as the box, and outliers as scatter plot symbols.
BW_IQR15_WHISKER_ALLPOINTS	Plot the minimum value within q25 - 1.5*IQR and maximum value within q75 + 1.5*IQR as the whiskers, the median, q25 and q75 values as the box, and plot all points as scatter plot symbols.

Where

q25	for a given population, q25 is the 25th percentile point in the population
q75	for a given population, q75 is the 75th percentile point in the population
IQR	for a given population, IQR is the value q75 - q25.



Turn on the numeric labeling of the Box and Whisker summary values (high whisker, q75, median, q25, low whisker) by setting the **BoxWhiskerPlot**. **ShowDatapointValue** property true. Turn on the numeric labeling of the scatter plot symbols used for outliers by setting the the **BoxWhiskerPlot**. **ScatterPlot.ShowDatapointValue** property true. You should not combine numeric labeling with the W_IQR15_WHISKER_ALLPOINTS option, unless you are working with a very small number of data points; otherwise the labels will all overlap one another.

Box and whisker plot example (extracted from the example program NewDemosRev2.BoxAndWhiskerChart)

[C#]

```
            //New York City
double[] NYCity = { 31.5, 33.6, 42.4, 52.5, 62.7, 71.6, 130, 76.8, 75.5, 68.2, 57.5,
47.6, 36.6 };
            //Houston
double[] Houston = { 50.4, 53.9, 60.6, 68.3, 74.5, 80.4, 5, 100, 82.6, 82.3, 78.2, 69.6,
61, 53.5 };
            //San Francisco
double[] SanFrancisco = { 48.7, 52.2, 53.3, 55.6, 58.1, 76, 61.5, 62.7, 63.7, 64.5, 61,
54.8, 49.4 };
            //Bouston
double[] Boston = { 32.4, 53.9, 44.6, 58.3, 64.5, 70.4, 73, 90, 72.6, 72.3, 68.2, 49.6,
41, 33.5 };
            //Pittsburgh
double[] Pittsburgh = { 41.4, 54, 24.6, 38.3, 44.5, 61.4, 63, 105, 72.6, 72.3, 68.2,
49.6, 41, 33.5 };
double minval = 0.0, maxval = 0.0;
```

```
int i;
int numpnts = NYCity.Length + Houston.Length + SanFrancisco.Length + Boston.Length +
Pittsburgh.Length;
.
.
.
ChartAttribute defaultattrib = new ChartAttribute(Colors.Black, 1, ChartObj.LS_SOLID,
Colors.Red);
defaultattrib.SetFillFlag(true);
ChartAttribute fillattrib = new ChartAttribute(Colors.Black, 3, ChartObj.LS_SOLID,
Colors.Red);
fillattrib.SetFillFlag(true);
BoxWhiskerPlot thePlot1 = new BoxWhiskerPlot(pTransform1, 0.25, fillattrib);
thePlot1.AddPopulation(NYCity, 1);
thePlot1.AddPopulation(Houston, 2);
thePlot1.AddPopulation(SanFrancisco, 3);
thePlot1.AddPopulation(Boston, 4);
thePlot1.AddPopulation(Pittsburgh, 5);
thePlot1.BWFormat = 1;
thePlot1.BarDatapointLabelPosition = ChartObj.CENTERED_BAR;
thePlot1.PlotLabelTemplate.TextFont = new ChartFont("Microsoft Sans Serif", 8,
FontStyle.Normal);
thePlot1.PlotLabelTemplate.DecimalPos = 1;
thePlot1.ShowDatapointValue = true;
// label outliers
thePlot1.ScatterPlot.ShowDatapointValue = true;
thePlot1.AutoBWChart();

chartVu.AddChartObject(thePlot1);
```

Bubble Plots

Class BubblePlot

GraphObj
|
+--ChartPlot
|
+--GroupPlot
|
+--BubblePlot

The **BubblePlot** class is a concrete implementation of the **GroupPlot** class. It displays bubble plots. A group dataset specifies the position and size of each bubble in a bubble plot. The numer of groups must be two.

BubblePlot constructor

```
[C#]
public BubblePlot(
   PhysicalCoordinates transform,
   GroupDataset dataset,
   int bubblesizetype,
   ChartAttribute attrib
);
```

transform The coordinate system for the new bubble plot object.

dataset	A group dataset specifying the location and size of the bubbles in the bubble plot. The number of groups must be two. The dataset values for X and Y[0] set the position of the center of each bubble and the values for Y[1] set the size of each bubble, either the area (SIZE_BUBBLE_AREA) or the radius(SIZE_BUBBLE_RADIUS).
bubblesizetype	Sets whether the circle representing each bubble plot has a radius, or an area, proportional to the Y[1] data values in the group dataset. Set using one of the bubble plot type constants: SIZE_BUBBLE_RADIUS or SIZE_BUBBLE_AREA.
attrib	Specifies the attributes (line color and fill color) of the bubble plot circles.

An individual bubble in a bubble plot object can have unique attributes. Use the objects **SetSegmentAttributesMode** and **SetSegmentAttributes** methods in the manner described for **SimplePlot** objects..

Bubble plot example (extracted from the example program ScatterPlots, class BubbleChart)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
TimeGroupDataset Dataset1 = new TimeGroupDataset("First",x1,y1);
Dataset1.SetStackMode(ChartObj.AUTOAXES_STACKED);
TimeCoordinates pTransform1 =
new TimeCoordinates( ChartObj.TIME_SCALE, ChartObj.LINEAR_SCALE);
pTransform1.AutoScale(Dataset1, ChartObj.AUTOAXES_FAR, ChartObj.AUTOAXES_FAR);
pTransform1.SetGraphBorderDiagonal(0.15, .15, .80, 0.75) ;
.
.  // Define axes, axes labels and grids
.
ChartAttribute attrib1 = new ChartAttribute (Colors.Black, 0,ChartObj.LS_SOLID);
attrib1.SetFillColor (Color.FromArgb(255,177, 33, 33));
attrib1.SetFillFlag (true);
BubblePlot thePlot1 = new BubblePlot(pTransform1, Dataset1,
                        ChartObj.SIZE_BUBBLE_RADIUS, attrib1);
chartVu.AddChartObject(thePlot1);
```

*Note the use of the **GroupDataset** method **SetStackMode**. This forces the auto-scale routine to look at the sum of y-values across groups, as is needed to auto-scale stacked plots. It is useful for bubble plots of type SIZE_BUBBLE_RADIUS because the y[0] value represents the y-position of the bubble, and the y[1] value the radius in physical coordinates. Adding the two for each bubble gives the maximum y-value for the scale needed to display the bubble. If SIZE_BUBBLE_AREA is used you may want to restrict the auto-scale routines to the just look

at the bubble position using **SetAutoScaleNumberGroups(1),** as seen in the **ArrowPlot** example. You could then add in some fudge factor to make sure that the scale shows the entire bubble. The example under CellPlot demonstrates this.

Candlestick Plots

Class CandlestickPlot

GraphObj
|
+--ChartPlot
|
+--GroupPlot
|
+-- CandlestickPlot

The **CandlestickPlot** class is a concrete implementation of the **GroupPlot** class. It extends the **GroupPlot** class and displays stock market data in an open-high-low-close format common in financial technical analysis. Every item of the plot is a group of two horizontal lines representing High and Low values which are connected with a vertical line and a box representing the Open and Close values. If the Open value is greater than the Close value for a particular candlestick, the box is filled, otherwise it is unfilled. The number of groups must be four. The data in the dataset is organized in the following manner: The Y[0] values of the group dataset represent the values for Open, the Y[1] values for High, the Y[2] values for Low, and the Y[3] values for Close.

CandlestickPlot constructor

```
[C#]
public CandlestickPlot(
   PhysicalCoordinates transform,
   GroupDataset dataset,
   double rwidth,
   ChartAttribute defaultattrib,
   ChartAttribute fillattrib
);
```

transform The coordinate system for the new **CandlestickPlot** object.

dataset The **CandlestickPlot** plot represents the group open-high-low-close values in this group dataset. The number of groups must be four. Orgainize the data in the following manner: The x-values of the group dataset set the x-positions of the candlestick objects. The Y[0] values of the group dataset represent the values for Open, the Y[1] values for High, the Y[2] values for Low, and the Y[3] values for Close.

rwidth	The width of the candlestick box in physical coordinates.
defaultattrib	Specifies the default attributes (line color and fill color) of the candlestick lines and box.
fillattrib	Specifies the attributes (line color and fill color) of the candlestick lines when the close value is greater than the open value.

An individual candlestick in a candlestick plot object can have unique attributes. Use the objects **SetSegmentAttributesMode** and **SetSegmentAttributes** methods in the manner described for **SimplePlot** objects..

Candlestick plot example (extracted from the example program FinancialExamples, class CandlestickChart)

[C#]

```
TimeGroupDataset Dataset1 =
       new TimeGroupDataset("Stock Data",xValues,stockPriceData);
.
.  // Define axes, axes labels and grids
.
ChartAttribute defaultattrib =
        new ChartAttribute(Colors.Black, 1,ChartObj.LS_SOLID, Colors.White);
defaultattrib.SetFillFlag(true);
ChartAttribute fillattrib =
       new ChartAttribute(Colors.Black, 1,ChartObj.LS_SOLID, Colors.Red);
fillattrib.SetFillFlag(true);
CandlestickPlot thePlot1 =
       new CandlestickPlot(pTransform1, Dataset1,
           ChartCalendar.GetCalendarWidthValue(ChartObj.DAY_OF_YEAR,0.8),
           defaultattrib, fillattrib);
```

* Note how the **ChartCalendar.GetCalendarWidthValue** method calculates the width of the bars as a function of time, in this case a width of 0.8 months.

Cell Plots

Class CellPlot

GraphObj

|

+--ChartPlot

|

+--GroupPlot

```
|
+--CellPlot
```

The **CellPlot** class extends the **GroupPlot** class and displays cell plots. A cell plot is a collection of rectangular objects with independent positions, widths and heights, specified using the values of the associated group dataset. The number of groups must be three. The (X, Y[0]) values of the group dataset represent the xy position of the lower left corner of each cell, the Y[1] values set the width of the cell, and the Y[2] values set the height of the cell. Each cell can be filled using a color, or an image.

CellPlot constructor

```
[C#]
public CellPlot(
   PhysicalCoordinates transform,
   GroupDataset dataset,
   ChartAttribute attrib
);
```

transform	The coordinate system for the new **CellPlot** object.
dataset	The cell plot represents the values in this group dataset. The number of groups must be three. The (X, Y[0]) values of the group dataset represent the xy position of the lower left corner of each cell, the Y[1] values set the width of the cell, and the Y[2] values set the height of the cell.
attrib	Specifies the attributes (line color and line style) for the cell plot.

Cells can be filled with an image instead of a solid color. Use the **CellPlot.SetPlotImage** method to place a **WriteableBitmap.BitmapImage** Image object in the cells of a cell plot. One image applies to all of the cells in the cell plot.

An individual cell in the cell plot object can have unique attributes. Use the objects **SetSegmentAttributesMode** and **SetSegmentAttributes** methods in the manner described for **SimplePlot** objects.

Cell plot example (extracted from the example program ScatterPlots, class CellPlotChart)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
```

```
GroupDataset Dataset1 = new GroupDataset("First",x1,y1);
Dataset1.SetAutoScaleNumberGroups(1); // picks up on width, but because data is
                                      // should still work
CartesianCoordinates pTransform1 =
       new CartesianCoordinates( ChartObj.LINEAR_SCALE, ChartObj.LINEAR_SCALE);
pTransform1.AutoScale(Dataset1, ChartObj.AUTOAXES_FAR, ChartObj.AUTOAXES_FAR);
// Add-in max width of cells
double maxx = pTransform1.GetScaleStopX() + 20;
// Add-in max height of cells
double maxy = pTransform1.GetScaleStopY() + 10;
pTransform1.SetScaleStopX(maxx);
pTransform1.SetScaleStopY(maxy);
// Re-auto-scale to produce rounded axis values.
pTransform1.AutoScale(ChartObj.AUTOAXES_NEAR, ChartObj.AUTOAXES_NEAR);
pTransform1.SetGraphBorderDiagonal(0.15, .15, .90, 0.75) ;
.
.  // Define axes, axes labels and grids
.
ChartAttribute attrib1 = new ChartAttribute (Colors.Blue, 1,ChartObj.LS_SOLID);
attrib1.SetFillColor (Colors.Blue);
attrib1.SetFillFlag (true);
CellPlot thePlot1 = new CellPlot(pTransform1, Dataset1,  attrib1);
for (i=0; i < numPoints; i++)
    thePlot1.SetSegmentColor(i, Color.FromArgb(255,(int)(x1[i]),
              (int) (y1[0,i]*2.0), (int) ((y1[1,i] + y1[2,i])* 7)));
chartVu.AddChartObject(thePlot1);
```

*Note the use of the **GroupDataset** method **SetAutoScaleNumberGroups**. This forces the auto-scale routine to look at just the first group of values, Y[0], because those are the only absolute position values. The maximum cell width and height are calculated and added to the initial scale. The auto-scale function is then rerun, producing a coordinate system that takes into account the widths and heights of the cells.

Error Bar Plots

Class ErrorBarPlot

GraphObj

|

+--ChartPlot

|

+--GroupPlot

|

+-- ErrorBarPlot

The **ErrorBarPlot** class extends the **GroupPlot** class and displays error bars. Error bars are two lines positioned around a data point to signify the statistical error associated with the data point. The number of groups must be two. The (X, Y[0]) values of the group dataset represent the xy position of the first error bar lines, the (X, Y[1]) values of the group dataset represent the xy position of the second error bar lines. The error bar lines center on the X. Connecting the error bar lines with a perpendicular line is an option.

ErrorBarPlot constructor

```
[C#]
public ErrorBarPlot(
   PhysicalCoordinates transform,
   GroupDataset dataset,
   double rbarwidth,
   ChartAttribute attrib
);
```

transform	The coordinate system for the new **ErrorBarPlot** object. The number of groups must be two.
dataset	The error bar plot represents the values in this group dataset. The number of groups must be two. The (X, Y[0]) values of the group dataset represent the xy position of the first error bar lines, the (X,Y[1]) values of the group dataset represent the xy position of the second error bar lines.
rbarwidth	The width of the error bars.
attrib	Specifies the attributes (line color and line style) for the error bars.

An individual set of error bars in an error bar plot object can have unique attributes. Use the objects **SetSegmentAttributesMode** and **SetSegmentAttributes** methods in the manner described for **SimplePlot** objects.

Floating Bar Plots

Class FloatingBarPlot

GraphObj
|
+--ChartPlot
|
+--GroupPlot
|
+-- FloatingBarPlot

The **FloatingBarPlot** class extends the **GroupBarPlot** class and displays floating bar plots. The bars are free floating because each bar does not reference a fixed base value, as do the simple bar plots, stacked bar plots and group bar plots. The number of groups must be two. The (X, Y[0])

values of the group dataset represent the starting points of each bar, the (X, Y[1]) values of the group dataset represent the ending points of each bar. All bars in a given **FloatingBarPlot** object have the same width.

FloatingBarPlot constructor

```
[C#]
public FloatingBarPlot(
   PhysicalCoordinates transform,
   GroupDataset dataset,
   double rbarwidth,
   ChartAttribute attrib,
   int nbarjust
);
```

transform	The coordinate system for the new **FloatingBarPlot** object.
dataset	The floating bar plot represents the values in this group dataset. The number of groups must be two. The (X, Y[0]) values of the group dataset represent the starting points of each bar, the (X, Y[1]) values of the group dataset represent the ending points of each bar.
rbarwidth	The width of the floating bars in units of the independent axis.
attrib	Specifies the attributes (line and fill color) for the floating bars.
nbarjust	Specifies the justification with respect to the independent data value. Use one of the justification constants: JUSTIFY_MIN, JUSTIFY_CENTER, JUSTIFY_MAX.

An individual bar in a floating bar plot object can have unique attributes. Use the objects **SetSegmentAttributesMode** and **SetSegmentAttributes** methods in the manner described for **SimplePlot** objects..

Scheduling charts often use floating bars, where the starting and ending values of the bar represent the duration of some aspect of a project. The default use of the floating bar class assumes that the ends of the bars are floating point values, not date/time values. Yet the scheduling chart often uses date/time values to specify the bar ends. Since only the x-axis works with time values, the floating bars of a scheduling chart need to used in the horizontal orientation mode, set using the **FloatingBar.SetBarOrient** method. Used in this mode, the x-values of the dataset position the bars with respect to the y-axis, and the y-values of the dataset position the ends of the bars with respect to the x-axis. In order to have a group dataset object that stores x-values as doubles and the y-group values as **ChartCalendar** dates you must use a special **TimeGroupDataset** constructor:

```
[C#]
public TimeGroupDataset(
   string sname,
   ChartCalendar[] x,
```

```
   ChartCalendar[,] y
);
```

If you manually scale the **TimeCoordinates** object, you can proceed as in all the other examples that use a date/time x-axis. If you need to use the auto-scaling capability, you will need to add a few additional steps.

Place the data in a **TimeGroupDataset** dataset and use it to auto-scale a **TimeCoordinates** scaling object. The only problem here is that since the y-values are the time values, the chart y-axis will end up as the time axis and the x-axis will end up the numeric axis. Call the **TimeCoordinates.SwapScaleOrientation** in order to get it back to the orientation we want. This swaps the scale orientation so the x-axis is the time axis and the y-axis is the numeric axis. See the second of the two examples below for more details about how to use date/time values to specify the bar ends of a floating bar plot.

Floating bar plot example that uses numeric values as the floating bar endpoints (extracted from the example program Bargraphs, class FloatingBars)

[C#]

```
GroupDataset Dataset1 =
   new GroupDataset("Actual Sales",x1,y1);
CartesianCoordinates pTransform1 = new CartesianCoordinates();
pTransform1.SetScaleStartX(0);
pTransform1.SetScaleStartY(0);
pTransform1.SetScaleStopX(12);
pTransform1.SetScaleStopY(7);
.
.  // Define axes, axes labels and grids
.
FloatingBarPlot thePlot1 = new FloatingBarPlot(pTransform1);
ChartAttribute attrib1 =
   new ChartAttribute (Colors.Black, 1,ChartObj.ChartObj.LS_SOLID, Colors.Green);
attrib1.SetFillFlag(true);
thePlot1.floatingBarPlot(Dataset1, 0.75, attrib1, ChartObj.JUSTIFY_CENTER);
thePlot1.SetBarOrient(ChartObj.HORIZ_DIR);
```

Floating bar plot example that uses date/time values as the floating bar endpoints (extracted from the example program CalendarData, class FloatingBars2s)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
int nnumpnts = 18;
int numgroups = 2;
```

```
double []x1= new double[nnumpnts];
ChartCalendar [,]y1 = new ChartCalendar[numgroups,nnumpnts];

x1[0] = 6;
y1[0,0] = new ChartCalendar(2002, ChartObj.JANUARY,1);
y1[1,0] = new ChartCalendar(2003, ChartObj.JANUARY,1);

x1[1] = 5;
y1[0,1] = new ChartCalendar(2002, ChartObj.JANUARY,1);
y1[1,1] = new ChartCalendar(2002, ChartObj.MAY,1);
.
.
.
x1[17] = 1;
y1[0,17] = new ChartCalendar(2002, ChartObj.JULY,1);
y1[1,17] = new ChartCalendar(2002, ChartObj.OCTOBER,1);

theFont = new ChartFont("Microsoft Sans Serif", 10,  FontStyle.Normal, FontWeights.Bold);
TimeGroupDataset Dataset1 = new TimeGroupDataset("Actual Sales",x1,y1);
TimeCoordinates pTransform1 = new TimeCoordinates();
pTransform1.AutoScale(Dataset1,ChartObj.AUTOAXES_FAR,  ChartObj.AUTOAXES_NEAR);

pTransform1.SwapScaleOrientation();
pTransform1.SetScaleStartY(0);
pTransform1.SetGraphBorderDiagonal(0.22, .15, .95, 0.8) ;
.
.  // Define axes, axes labels and grids
.
FloatingBarPlot thePlot1 = new FloatingBarPlot(pTransform1);
ChartAttribute attrib1 =
       new ChartAttribute (Colors.Black, 1,ChartObj.LS_SOLID, Colors.Green);
attrib1.SetFillFlag(true);
thePlot1.InitFloatingBarPlot(Dataset1, 0.75, attrib1, ChartObj.JUSTIFY_CENTER);
thePlot1.SetBarOrient(ChartObj.HORIZ_DIR);
chartVu.AddChartObject(thePlot1);
```

Floating Stacked Bar Plots

Class FloatingStackedBarPlot

GraphObj

|

+--ChartPlot

|

+--GroupPlot

|

+-- FloatingStackedBarPlot

The **FloatingStackedBarPlot** class extends the **GroupPlot** class and displays floating stacked bar plots. The bars are free floating because each bar does not reference a fixed base value, as do the simple bar plots, stacked bar plots and group bar plots. The starting value for each stacked bar is Y[0]. Each bar after that is defined by the succeeding value in the group value array (Y[1], Y[2]..). Unlike the **StackedBarPlot** plot, the displayed values are not a cumulative sum of the group values. All bars in a given **FloatingStackedBarPlot** object have the same width.

FloatingStackedBarPlot constructor

```
[C#]
public FloatingStackedBarPlot(
   PhysicalCoordinates transform,
   GroupDataset dataset,
   double rbarwidth,
   ChartAttribute[] attrib,
   int nbarjust
);
```

transform	The coordinate system for the new **FloatingStackedBarPlot** object.
dataset	The starting value for each stacked bar is Y[0]. Each bar after that is defined by the succeeding value in the group value array (Y[1], Y[2]..).
rbarwidth	The width of the floating bars in units of the independent axis.
attrib	An array of **ChartAttribute** specifying the color for each bar. The number of colors, and the length of the array should be one less than the number of groups in the source dataset..
nbarjust	Specifies the justification with respect to the independent data value. Use one of the justification constants: JUSTIFY_MIN, JUSTIFY_CENTER, JUSTIFY_MAX.

FloatingStackedBarPlot plots can be used in place of **OHLCPlots**, or **CandlestickPlots** for the display of financial information. See the example below.

Floating stacked bar plot example for displaying stock data. Extracted from the NewDemosRev2. FloatingStackedBars example.

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
TimeGroupDataset Dataset1 = new TimeGroupDataset("Stock Data",xValues,stockPriceData);
pTransform1 = new TimeCoordinates();
pTransform1.SetWeekType(weekmode);
pTransform1.AutoScale(Dataset1,ChartObj.AUTOAXES_NEAR,  ChartObj.AUTOAXES_NEAR);
pTransform1.SetGraphBorderDiagonal(0.1, .15, .90, 0.75) ;

SetInitialDates(pTransform1);

Background graphbackground1 =
    new Background( pTransform1, ChartObj.GRAPH_BACKGROUND,
    Colors.White, Colors.LightGray, ChartObj.Y_AXIS);
```

```
chartVu.AddChartObject(graphbackground1);
Background plotbackground1 =
    new Background( pTransform1, ChartObj.PLOT_BACKGROUND,Colors.White);
chartVu.AddChartObject(plotbackground1);
.
.
.
ChartAttribute attrib1 =
    new ChartAttribute(Colors.Red, 1, ChartObj.LS_SOLID, Colors.Red);
ChartAttribute attrib2 =
    new ChartAttribute(Colors.Yellow, 1, ChartObj.LS_SOLID, Colors.Yellow);
ChartAttribute attrib3 =
    new ChartAttribute(Colors.Blue, 1, ChartObj.LS_SOLID, Colors.Blue);
ChartAttribute attrib4 =
    new ChartAttribute(Colors.Green, 1, ChartObj.LS_SOLID, Colors.Green);
ChartAttribute[] attribArray = { attrib1, attrib2, attrib3, attrib4 };
attrib1.SetFillFlag(true);
thePlot1 = new FloatingStackedBarPlot(pTransform1,
    Dataset1, ChartCalendar.GetCalendarWidthValue(ChartObj.DAY_OF_YEAR, 0.75),
    attribArray, ChartObj.JUSTIFY_CENTER);
thePlot1.SetFastClipMode(ChartObj.FASTCLIP_X);
chartVu.AddChartObject(thePlot1);
```

Group Bar Plots

Class GroupBarPlot

GraphObj
|
+--ChartPlot
|
+--GroupPlot
|
+--GroupBarPlot

The **GroupBarPlot** class extends the **GroupPlot** class and displays data in a group bar format. Individual bars, the height of which corresponds to the group values (Y[0], Y[1], Y[2], ...) of the dataset, are displayed side by side, as a group, justified with respect to the X-position value for each group.

GroupBarPlot constructor

```
[C#]
public GroupBarPlot(
   PhysicalCoordinates transform,
   GroupDataset dataset,
   double rbarwidth,
   double rbarbase,
   ChartAttribute[] attribs,
   int nbarjust
);
```

transform The coordinate system for the new **GroupPlot** object.

dataset The group bar graph represents the values in this group dataset. Individual bars, the height of which corresponds to the group values (Y[0], Y[1], Y[2], ...) of the dataset.

rbarwidth The width of the group bars in units of the independent axis. All bars within a group are squeezed into the width defined by *rbarwidth*. Each individual bar within the group has a width of *rbarwidth*/*dataset*.GetNumberGroups().

rbarbase The group bars start at the value *rbarbase*, and extend to the group bar values represented by the dataset.

attribs An array of **ChartAttribute** objects, sized the same as the number of groups in the dataset specify the attributes (outline color and fill color) for each group of a group bar graph.

nbarjust The group bars are justified with respect to the x-values in the dataset using the *rbarjust* justification value (JUSTIFY_MIN, JUSTIFY_CENTER, or JUSTIFY_MAX).

The attributes for each group can set or modified using the SetSegment… methods, where the segment number parameter corresponds to the group number. These methods include SetSegmentAttributes, SetSegmentFillColor, SetSegmentLineColor, and SetSegmentColors.

Group bar plot example (extracted from the example program Bargraphs, class GroupBargraphs)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
TimeGroupDataset Dataset1 =
        new TimeGroupDataset("GroupTimeData",xValues,groupBarData);
TimeCoordinates pTransform1 = new TimeCoordinates();
pTransform1.AutoScale(Dataset1,ChartObj.AUTOAXES_NEAR,  ChartObj.AUTOAXES_NEAR);
pTransform1.SetTimeScaleStart(new ChartCalendar(1997,ChartObj.JANUARY,1));
pTransform1.SetTimeScaleStop(new ChartCalendar(2003,ChartObj.JANUARY,1));
pTransform1.SetGraphBorderDiagonal(0.1, .1, .45, 0.75) ;
Background background1 = new Background( pTransform1, ChartObj.GRAPH_BACKGROUND,
           Color.FromArgb(255,0,120,70), Color.FromArgb(255,0,40,30), ChartObj.Y_AXIS);
chartVu.AddChartObject(background1);
.
.  // Define axes, axes labels and grids
.
ChartAttribute attrib1 =
           new ChartAttribute(Colors.Red, 1,ChartObj.LS_SOLID, Colors.Red);
ChartAttribute attrib2 =
           new ChartAttribute(Colors.Yellow, 1,ChartObj.LS_SOLID, Colors.Yellow);
ChartAttribute attrib3 =
```

```
            new ChartAttribute(Colors.Blue, 1,ChartObj.LS_SOLID, Colors.Blue);
ChartAttribute attrib4 =
            new ChartAttribute(Colors.Green, 1,ChartObj.LS_SOLID, Colors.Green);
ChartAttribute []attribArray = {attrib1, attrib2, attrib3, attrib4};
GroupBarPlot thePlot1 = new GroupBarPlot(pTransform1, Dataset1,
            ChartCalendar.GetCalendarWidthValue(ChartObj.YEAR,0.75), 0.0,
            attribArray, ChartObj.JUSTIFY_CENTER);
thePlot1.SetBarOverlap(0.0);
chartVu.AddChartObject(thePlot1);
```

Histogram Plots

Class HistogramPlot

GraphObj
|
+--ChartPlot
|
+--GroupPlot
|
+--HistogramPlot

The **HistogramPlot** class extends the **GroupPlot** class and displays histogram plots. A histogram plot is a collection of rectangular objects with independent positions, widths and heights, specified using the values of the associated group dataset. The number of groups must be two. The X-values of the group dataset represent the x-position of the lower left corner of each histogram bar, the Y[0] values set the height of each histogram bar, and the Y[1] values set the width of each histogram bar. The histogram bars share a common base value.

Histogram constructor

```
[C#]
public HistogramPlot(
   PhysicalCoordinates transform,
   GroupDataset dataset,
   double rbarbase,
   ChartAttribute attrib
);
```

transform — The coordinate system for the new **HistogramPlot** object.

dataset — The histogram plot represents the values in this group dataset. The number of groups must be two. The X-values of the group dataset represent the x-position of the lower left corner of each histogram bar, the Y[0] values set the height of each histogram bar, and the Y[1] values set the width of each histogram bar.

rbarbase — The histogram bars start at the value *rbarbase*, and extend to the histogram bar values represented by the dataset.

attrib Specifies the attributes (line color and line style) for the histogram bars.

An individual histogram bar in the histogram plot object can have unique attributes. Use the objects **SetSegmentAttributesMode** and **SetSegmentAttributes** methods in the manner described for **SimplePlot** objects. Each histogram bar can be labeled with the Y[0] group value bar (bar height) using the bar data point methods, see the example below.

Histogram plot example (extracted from the example program Bargraphs, class HistogramBars)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
int nnumpnts = 6;
int numgroups = 2;
double []x1= new double[nnumpnts];
double [,]y1 = new double[numgroups,nnumpnts];

//            height           width
x1[0] = 0;  y1[0,0] = .12; y1[1,0] = 13;
x1[1] = 13; y1[0,1] = .97; y1[1,1] = 7;
x1[2] = 20; y1[0,2] = .80; y1[1,2] = 10;
x1[3] = 30; y1[0,3] = .44; y1[1,3] = 10;
x1[4] = 40; y1[0,4] = .28; y1[1,4] = 20;
x1[5] = 60; y1[0,5] = .4; y1[1,5] = 20;

theFont = new ChartFont("Microsoft Sans Serif", 10,  FontStyle.Normal, FontWeights.Bold);

GroupDataset Dataset1 = new GroupDataset("Actual Sales",x1,y1);
CartesianCoordinates pTransform1 = new CartesianCoordinates();

pTransform1.SetScaleStartY(0);
pTransform1.SetScaleStartX(0);
pTransform1.SetScaleStopX(80);
pTransform1.SetScaleStopY(1.00);
pTransform1.SetGraphBorderDiagonal(0.15, .15, .9, 0.75) ;
Background graphbackground =
            new Background( pTransform1, ChartObj.GRAPH_BACKGROUND,
                Color.FromArgb(255,30,70,70), Color.FromArgb(255,90,20,155),
                ChartObj.Y_AXIS);
chartVu.AddChartObject(graphbackground);

Background plotbackground = new Background( pTransform1,
            ChartObj.PLOT_BACKGROUND, Colors.Black);
chartVu.AddChartObject(plotbackground);
.
.  // Define axes, axes labels and grids
.
ChartAttribute attrib1 =
            new ChartAttribute (Colors.Black, 0,ChartObj.LS_SOLID, Colors.Green);
attrib1.SetFillFlag(true);
HistogramPlot thePlot1 = new HistogramPlot(pTransform1, Dataset1, 0.0, attrib1);

NumericLabel bardatavalue = thePlot1.GetPlotLabelTemplate();
```

```
bardatavalue.SetTextFont(theFont);
bardatavalue.SetNumericFormat(ChartObj.PERCENTFORMAT);
bardatavalue.SetColor(Colors.Black);
thePlot1.SetBarDatapointLabelPosition(ChartObj.INSIDE_BAR);
thePlot1.SetPlotLabelTemplate(bardatavalue);
thePlot1.SetShowDatapointValue(true);

thePlot1.SetSegmentAttributesMode(true);
thePlot1.SetSegmentFillColor(0,Colors.Red);
thePlot1.SetSegmentFillColor(1, Colors.Magenta);
thePlot1.SetSegmentFillColor(2, Colors.Blue);
thePlot1.SetSegmentFillColor(3, Colors.Green);
thePlot1.SetSegmentFillColor(4, Colors.Yellow);
thePlot1.SetSegmentFillColor(5, Colors.Pink);
chartVu.AddChartObject(thePlot1);
```

Line Gap Plots

Class LineGapPlot

GraphObj
|
+--ChartPlot
|
+--GroupPlot
|
+-- LineGapPlot

The **LineGapPlot** class extends the **GroupPlot** class and displays a line gap chart. The number of groups must be two. A line gap chart consists of two line plots where a contrasting color fills and highlights the area between the two lines. The (X, Y[0]) values of the group dataset represent the first of the bounding lines, and the (X,Y[1]) values of the group dataset represent the second of the bounding lines.

LineGapPlot constructor

```
[C#]
public LineGapPlot(
   PhysicalCoordinates transform,
   GroupDataset dataset,
   ChartAttribute attrib
);
```

transform	The coordinate system for the new **LineGapPlot** object.
dataset	The line gap plot represents the values in this group dataset. The number of groups in this group dataset must be two.
attrib	Specifies the attributes (line and fill color) for the fill area.

A segment between adjacent x-values in the line gap plot object can have unique attributes. Use the objects **SetSegmentAttributesMode** and **SetSegmentAttributes** methods in the manner described for **SimplePlot** objects..

Line gap bar plot example (extracted from the example program MiscCharts, class LineGapChart)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
int nNumPnts = 5, nNumGroups = 2;
ChartCalendar []xValues=  new ChartCalendar[nNumPnts];
double [,]groupBarData = new double[nNumGroups,nNumPnts];

theFont = new ChartFont("Microsoft Sans Serif", 10,  FontStyle.Normal, FontWeights.Bold);
xValues[0] = new ChartCalendar(1998, ChartObj.JANUARY, 1);
groupBarData[0,0] = 43;  groupBarData[1,0] = 71;

xValues[1] = new ChartCalendar(1999, ChartObj.JANUARY, 1);
groupBarData[0,1] = 40;  groupBarData[1,1] = 81;

xValues[2] = new ChartCalendar(2000, ChartObj.JANUARY, 1);
groupBarData[0,2] = 54;  groupBarData[1,2] = 48;

xValues[3] = new ChartCalendar(2001, ChartObj.JANUARY, 1);
groupBarData[0,3] = 56;  groupBarData[1,3] = 44;

xValues[4] = new ChartCalendar(2002, ChartObj.JANUARY, 1);
groupBarData[0,4] = 58;  groupBarData[1,4] = 40;

TimeGroupDataset Dataset1 =
       new TimeGroupDataset("GroupTimeData",xValues,groupBarData);

TimeCoordinates pTransform1 = new TimeCoordinates();
pTransform1.AutoScale(Dataset1,ChartObj.AUTOAXES_NEAR,  ChartObj.AUTOAXES_FAR);

pTransform1.SetGraphBorderDiagonal(0.15, .1, .95, 0.8) ;
Background background = new Background( pTransform1, ChartObj.GRAPH_BACKGROUND,
Color.FromArgb(255,0,120,70), Color.FromArgb(255,0,40,30), ChartObj.Y_AXIS);
chartVu.AddChartObject(background);
.
.  // Define axes, axes labels and grids
.
ChartAttribute attrib1 =
       new ChartAttribute(Colors.Black, 1,ChartObj.LS_SOLID, Colors.Red);
attrib1.SetFillFlag(true);
attrib1.SetLineFlag(false);
LineGapPlot thePlot1 = new LineGapPlot(pTransform1, Dataset1, attrib1);
chartVu.AddChartObject(thePlot1);
```

Multi-Line Plots

Class MultiLinePlot

GraphObj
|
+--ChartPlot
|
+--GroupPlot
|
+-- MultiLinePlot

The **MultiLinePlot** class extends the **GroupPlot** class and displays group data in multi-line format. A group dataset with eight groups will display eight separate line plots. The y-values for each group of the dataset are the y-values for each line in the plot. Each line plot share the same x-values of the group dataset.

MultiLinePlot constructor

```
[C#]
public MultiLinePlot(
   PhysicalCoordinates transform,
   GroupDataset dataset,
   ChartAttribute[] attribs
);
```

transform	The coordinate system for the new **MultiLinePlot** object.
dataset	The multi-line plot represents the values in this group dataset.
attribs	An array of **ChartAttribute** objects, sized the same as the number of groups in the dataset,t specify the attributes (line color and line style) for each group of the multi-line plot.

The attributes for each group can set or modified using the SetSegment… methods, where the segment number parameter corresponds to the group number. These methods include SetSegmentAttributes, SetSegmentFillColor, SetSegmentLineColor, and SetSegmentColors.

Multi-line plot example (extracted from the example program MultiLinePlots, class MultiLine)

[C#]

```
public void InitializeChart(ChartView chartVu)
```

```
{
.
.
.
int  numPoints = 100;
int numGroups = 7;
double []x1 = new double[numPoints];
double [,]y1 = new double[numGroups,numPoints];
int i, j;

for (i=0; i < numPoints; i++)
{
    x1[i] =  (double)i * 0.2;
    for (j = 0; j < numGroups; j++)
    y1[j,i] = j * (i * 0.01) + (double)(j+1) * 5.0 * (1.0 - Math.Exp(-x1[i]/0.7));
}
GroupDataset Dataset1 = new GroupDataset("First",x1,y1);

CartesianCoordinates pTransform1 =
       new CartesianCoordinates( ChartObj.LINEAR_SCALE, ChartObj.LINEAR_SCALE);
pTransform1.AutoScale(Dataset1, ChartObj.AUTOAXES_FAR, ChartObj.AUTOAXES_FAR);

pTransform1.SetScaleStartX(0);
pTransform1.SetScaleStartY(0);

Background background = new Background( pTransform1, ChartObj.PLOT_BACKGROUND,
              Color.FromArgb(255,255,255,255));
chartVu.AddChartObject(background);
pTransform1.SetGraphBorderDiagonal(0.15, .15, .90, 0.70) ;
.
.       // Define axes, axes labels and grids
.
ChartAttribute attrib1 = new ChartAttribute (Colors.Blue, 3,ChartObj.LS_SOLID);
ChartAttribute []attribArray = new ChartAttribute[numGroups];
for (i=0; i < numGroups; i++)
    attribArray[i] = (ChartAttribute) attrib1.Clone();
MultiLinePlot thePlot1 = new MultiLinePlot(pTransform1, Dataset1,  attribArray);
chartVu.AddChartObject(thePlot1);
```

Open-High-Low-Close Plots

Class OHLCPlot

GraphObj

|

+--ChartPlot

|

+--GroupPlot

|

+-- OHLCPlot

The **OHLCPlot** class extends the **GroupPlot** class and displays stock market data in an open-high-low-close format common in financial technical analysis. Every item of the plot is a vertical line, representing High and Low values, with two small horizontal "flags", one left and one right extending from the vertical High-Low line and representing the Open and Close values. The number of groups must be four. The Y[0] values of the group dataset represent the values for Open, the Y[1] values for High, the Y[2] values for Low, and the Y[3] values for Close.

OHLCPlot constructor

```
[C#]
public OHLCPlot(
   PhysicalCoordinates transform,
   GroupDataset dataset,
   double rflagwidth,
   ChartAttribute attrib
);
```

transform	The coordinate system for the new **OHLCPlot** object.
dataset	The **OHLCPlot** plot will represent the group open-high-low-close values in this group dataset. The number of groups must be four. The Y[0] values of the group dataset represent the values for Open, the Y[1] values for High, the Y[2] values for Low, and the Y[3] values for Close.
rflagwidth	The width of the open and close markers in units of the independent axis.
attrib	Specifies the attributes (line color and line style) for the open-high-low-close plot.

An individual OHLC element in an OHLC plot object can have unique attributes. Use the objects **SetSegmentAttributesMode** and **SetSegmentAttributes** methods in the manner described for **SimplePlot** objects.

OHLC plot example (extracted from the example program FinancialExamples, class OHLCChart)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
TimeGroupDataset Dataset1 =
         new TimeGroupDataset("Stock Data",xValues,
                              stockPriceData);
TimeCoordinates pTransform1 = new TimeCoordinates();
pTransform1.SetWeekType (ChartObj.WEEK_5D);
pTransform1.AutoScale(Dataset1, ChartObj.AUTOAXES_NEAR,
                              ChartObj.AUTOAXES_NEAR);.
.
.       // Define axes, axes labels and grids
.
ChartAttribute attrib1 =
       new ChartAttribute(Colors.Red, 1,ChartObj.ChartObj.LS_SOLID, Colors.Red);
attrib1.SetFillFlag(true);
OHLCPlot thePlot1 = new OHLCPlot(pTransform1, Dataset1,
               ChartCalendar.GetCalendarWidthValue(ChartObj.DAY_OF_YEAR,0.75),
               attrib1);
chartVu.AddChartObject(thePlot1);
```

* Note how the **ChartCalendar.GetCalendarWidthValue** method calculates the width of the bars as a function of time, in this case a width of 0.75 days.

Stacked Bar Plots

Class StackedBarPlot

GraphObj
|
+--ChartPlot
|
+--GroupPlot
|
+--StackedBarPlot

The **StackedBarPlot** class extends the **GroupPlot** class and displays data in stacked bar format. In a stacked bar plot each group is stacked on top of one another, each group bar a cumulative sum of the related group items before it.

StackedBarPlot constructor

```
[C#]
public StackedBarPlot(
   PhysicalCoordinates transform,
   GroupDataset dataset,
   double rbarwidth,
   double rbarbase,
   ChartAttribute[] attribs,
   int nbarjust
);
```

transform	The coordinate system for the new **StackedBarPlot** object.
dataset	The stacked bar graph represents the values in this group dataset.
rbarwidth	The width of the stacked bars in units of the independent axis.
rbarbase	The stacked bars start at the value *rbarbase*, and extend to the group bar values represented by the dataset.
attribs	An array of **ChartAttribute** objects, sized the same as the number of groups in the dataset, that specify the attributes (outline color and fill color) for each group of a stacked bar graph.

nbarjust	The stacked bars are justified with respect to the x-values in the dataset using the *rbarjust* justification value (JUSTIFY_MIN, JUSTIFY_CENTER, or JUSTIFY_MAX).

Each stacked bar can be labeled with the group value bar using the bar data point methods, see the example below.

The attributes for each group can set or modified using the SetSegment… methods, where the segment number parameter corresponds to the group number. These methods include SetSegmentAttributes, SetSegmentFillColor, SetSegmentLineColor, and SetSegmentColors.

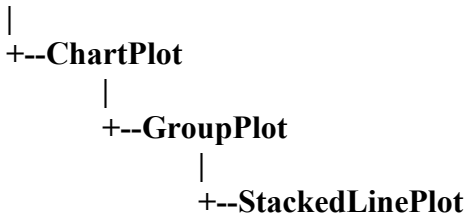
Stacked bar plot example (extracted from the example program Bargraphs, class GroupBargraphs)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
TimeCoordinates pTransform2 = new TimeCoordinates();
// User same dataset as Group bar plot, set stacked mode flag
Dataset1.SetStackMode(ChartObj.AUTOAXES_STACKED);
pTransform2.AutoScale(Dataset1,ChartObj.AUTOAXES_NEAR,  ChartObj.AUTOAXES_NEAR);

pTransform2.SetTimeScaleStart(new ChartCalendar(1997,ChartObj.JANUARY,1));
pTransform2.SetTimeScaleStop(new ChartCalendar(2003,ChartObj.JANUARY,1));
pTransform2.SetGraphBorderDiagonal(0.55, .1, .95, 0.75) ;

pTransform2.SetScaleStartY(0);
.
.   // Define axes, axes labels, and grids
.
StackedBarPlot thePlot2 =
           new StackedBarPlot(pTransform2, Dataset1,
           ChartCalendar.GetCalendarWidthValue(ChartObj.YEAR,0.75), 0.0,
           attribArray, ChartObj.JUSTIFY_CENTER);
NumericLabel bardatavalue = thePlot2.GetPlotLabelTemplate();
bardatavalue.SetTextFont(theFont);
bardatavalue.SetNumericFormat(ChartObj.CURRENCYFORMAT);
bardatavalue.SetDecimalPos(1);
bardatavalue.SetColor(Colors.Black);
thePlot2.SetPlotLabelTemplate(bardatavalue);
thePlot2.SetBarDatapointLabelPosition(ChartObj.CENTERED_BAR);
thePlot2.SetShowDatapointValue(true);
chartVu.AddChartObject(thePlot2);
```

Stacked Line Plots

Class StackedLinePlot

GraphObj


The **StackedLinePlot** class extends the **GroupPlot** class and displays data in stacked line format. In a stacked line plot each group is stacked on top of one another, each group line a cumulative sum of the groups before it.

StackedLinePlot constructor

```
[C#]
public StackedLinePlot(
   PhysicalCoordinates transform,
   GroupDataset dataset,
   ChartAttribute[] attribs
);
```

transform	The coordinate system for the new **StackedLinePlot** object.
dataset	The stacked line plot represents the values in this group dataset.
attribs	An array of **ChartAttribute** objects, sized the same as the number of groups in the dataset specify the attributes (line color and line style) for each group of the stacked line graph.

The attributes for each group can set or modified using the SetSegment… methods, where the segment number parameter corresponds to the group number. These methods include SetSegmentAttributes, SetSegmentFillColor, SetSegmentLineColor, and SetSegmentColors.

Stacked line plot example (extracted from the example program MultiLinePlots, class StackedLines)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
```

```
.
.
int  numPoints = 100;
int numGroups = 7;
double []x1 = new double[numPoints];
double [,]y1 = new double[numGroups,numPoints];
.
.  // Initialize data
.
theFont = new ChartFont("Microsoft Sans Serif", 10, FontStyle.Normal, FontWeights.Bold);

GroupDataset Dataset1 = new GroupDataset("First",x1,y1);
Dataset1.SetStackMode(ChartObj.AUTOAXES_STACKED);
CartesianCoordinates pTransform1 =
       new CartesianCoordinates( ChartObj.LINEAR_SCALE, ChartObj.LINEAR_SCALE);
pTransform1.AutoScale(Dataset1, ChartObj.AUTOAXES_FAR, ChartObj.AUTOAXES_FAR);
pTransform1.SetScaleStartX(0);
pTransform1.SetScaleStartY(0);

Background background = new Background( pTransform1, ChartObj.PLOT_BACKGROUND,
           Color.FromArgb(255,255,255,255));
chartVu.AddChartObject(background);

pTransform1.SetGraphBorderDiagonal(0.15, .15, .90, 0.75) ;
.
. // Define axes, axes labels and grids
.

ChartAttribute attrib1 = new ChartAttribute (Colors.Blue, 1,ChartObj.LS_SOLID);
attrib1.SetFillFlag(true);
attrib1.SetLineFlag(false);
ChartAttribute []attribArray = new ChartAttribute[numGroups];
for (i=0; i < numGroups; i++)
    attribArray[i] = (ChartAttribute) attrib1.Clone();
attribArray[0].SetFillColor(Colors.Blue);
attribArray[1].SetFillColor(Colors.Yellow);
attribArray[2].SetFillColor(Colors.Magenta);
attribArray[3].SetFillColor(Colors.Orange);
attribArray[4].SetFillColor(Colors.Gray);
attribArray[5].SetFillColor(Colors.Red);
attribArray[6].SetFillColor(Colors.Green);
StackedLinePlot thePlot1 =
       new StackedLinePlot(pTransform1, Dataset1,  attribArray);
chartVu.AddChartObject(thePlot1);
```

12. Contour Plotting

ChartPlot
ContourPlot

The **ContourPlot** class displays contour data organized in a **ContourDataset**.dataset. The line contour graph draws contituous lines through the data at xy-values representing equal values of z, analogous to the equal pressure lines (isobars) of a weather map. A filled contor graph fills the area between two contour levels with a specific color.

Line and Filled Contour Plots

Class ContourPlot

GraphObj
|
+--ChartPlot
|
+--ContourPlot

The **ContourPlot** class is a concrete implementation of the **ChartPlot** class and displays a contour plot using either lines, or regions filled with color. The two constructors below differ only by the inclusion of the contour line flags and the contour label flags in the second constructor.

ContourPlot constructors

```
[C#]
public ContourPlot(
   PhysicalCoordinates transform,
   ContourDataset dataset,
   double[] contourlevels,
   ChartAttribute[] attribs,
   int numcontourlevels,
   int contourtype
);
public ContourPlot(
   PhysicalCoordinates transform,
   ContourDataset dataset,
   double[] contourlevels,
   ChartAttribute[] attribs,
   bool[] blineflags,
   bool[] blabelflags,
   int numcontourlevels,
   int contourtype
);
```

transform	The coordinate system for the new **ContourPlot** object.
dataset	The **ContourDataset** plot will represent the xyz values in this contour data set.
contourlevels	An array, size [numcontourlevels], of the contour levels used in the contour plot.
attribs	An array of color and fill attributes, size [numcontourlevels+1]. If the *contourtype* is CONTOUR_LINE, the colors of elements 0..*numcontourlevels*-1 set the colors of the contour lines. If the contourtype is CONTOUR_FILL, elements 0..*numcontourlevels* set the colors of the contour regions.
blineflags	An array, size [numcontourlevels], of boolean flags specifying whether a contour line should be displayed.
blabelflags	An array, size [numcontourlevels], of boolean flags specifying whether a contour line should be labeled with the numeric value of the associated contour level.
numcontourlevels	The number of contour levels.
contourtype	Specifies if the contour plot uses contour lines (CONTOUR_LINE), filled contour regions (CONTOUR_FILL) or both (CONTOUR_LINEANDFILL)..

Contour plots that use the CONTOUR_FILL algorithm require one more color than the CONTOUR_LINE algorithm.

The attributes for each contour line can set or modified using the SetSegment… methods, where the segment number parameter corresponds to the contour value index.. These methods include SetSegmentAttributes, SetSegmentFillColor, SetSegmentLineColor, and SetSegmentColors.

Contour line plot example (extracted from the example program ContourPlots, class ContourLinePlot)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
double []contourlevels = { 1000, 1200, 1400, 1600, 1800,
                           1900, 2000, 2100, 2200, 2400, 2600, 2800, 3000};
```

```
int i;
theFont = new ChartFont("Microsoft Sans Serif", 10,  FontStyle.Normal, FontWeights.Bold);


CartesianCoordinates pTransform1 = new CartesianCoordinates(-7, -7, 7, 7);

CreateRegularGridPolysurface();
pTransform1.SetGraphBorderDiagonal(0.10, .10, .85, 0.85) ;
Background background =
        new Background( pTransform1, ChartObj.GRAPH_BACKGROUND, Colors.White);
chartVu.AddChartObject(background);

ChartText checkBoxCaption =
   new ChartText(pTransform1, theFont,"Contour Level", 560, 35, ChartObj.DEV_POS);
chartVu.AddChartObject(checkBoxCaption);

LinearAxis xAxis = new LinearAxis(pTransform1, ChartObj.X_AXIS);
chartVu.AddChartObject(xAxis);
LinearAxis yAxis = new LinearAxis(pTransform1, ChartObj.Y_AXIS);
chartVu.AddChartObject(yAxis);
NumericAxisLabels xAxisLab = new NumericAxisLabels(xAxis );
chartVu.AddChartObject(xAxisLab);
NumericAxisLabels yAxisLab = new NumericAxisLabels(yAxis);
chartVu.AddChartObject(yAxisLab);
ChartGrid xgrid = new ChartGrid(xAxis, yAxis,ChartObj.X_AXIS, ChartObj.GRID_MAJOR);
chartVu.AddChartObject(xgrid);

ChartGrid ygrid = new ChartGrid(xAxis, yAxis,ChartObj.Y_AXIS, ChartObj.GRID_MAJOR);
chartVu.AddChartObject(ygrid);

ChartAttribute []attribs = new ChartAttribute[numcontourlevels+1];
for (i=0; i <= numcontourlevels; i++)
{
    Color color =  Color.FromArgb(255,(int) (255* ChartSupport.GetRandomDouble()),
                      (int) (255 * ChartSupport.GetRandomDouble()), (int) (255 *
    ChartSupport.GetRandomDouble()));
     attribs[i] = new ChartAttribute(color,2,ChartObj.LS_SOLID,color);
     attribs[i].SetFillFlag(true);
}
attribs[0].SetColor(Colors.Black);
attribs[1].SetColor(Colors.Blue);
attribs[2].SetColor(Colors.DarkGray);
attribs[3].SetColor(Colors.Green);
attribs[4].SetColor(Colors.Red);
attribs[5].SetColor(Colors.Cyan);
attribs[6].SetColor(Colors.Magenta);
attribs[7].SetColor(Colors.Orange);
attribs[8].SetColor(Colors.Yellow);

for (i=0; i < numcontourlevels; i++)
{
   if ((i % 3) == 0)
       lineflags[i] = true;
    else
       lineflags[i] = false;
    if ((i % 3) == 0)
       labelflags[i] = true;
    else
       labelflags[i] = false;
}

thePlot1 = new ContourPlot(pTransform1, dataset1, contourlevels, attribs,
                  lineflags, labelflags, numcontourlevels, ChartObj.CONTOUR_LINE);
thePlot1.SetPolygonGridOn(true);
thePlot1.SetContourLineAlgorithm(ChartObj.CONTOUR_LINEWALK);
NumericLabel contourlabel = thePlot1.GetPlotLabelTemplate();
ChartFont contourLabelFont = new ChartFont("Microsoft Sans Serif", 8, FontStyle.Normal);
contourlabel.SetDecimalPos(0);
contourlabel.SetTextFont(contourLabelFont);
contourlabel.TextBgMode = true;
```

```
contourlabel.TextBgColor = Colors.White;
thePlot1.SetPlotLabelTemplate(contourlabel);
chartVu.AddChartObject(thePlot1);
```

13. Data Markers and Data Cursors

Marker
DataCursor

Data markers are symbols and lines that can be "dropped" on to the data presented in a graph, much like a bookmark in a word processing document. Place the markers in a chart under program control or in response to a mouse event in the graph window.

Data cursors are temporary lines or symbols, that are used to help position the mouse cursor over the desired section of a graph. Standard data cursors include cross hairs, a box, and horizontal and/or vertical lines.

Data Markers

Class Marker

GraphObj
|
+--Marker

Create data markers using the **Marker** class. The constructor below creates a new **Marker** object using the specified coordinate system, marker type, marker position and marker size.

Marker constructor

```
[C#]
public Marker(
   PhysicalCoordinates transform,
   int nmarkertype,
   double x,
   double y,
   double rsize,
   int npostype
);
```

transform	Places the marker in the coordinate system defined by transform.
nmarkertype	Specifies the shape of the current chart marker. Use one of the chart marker constants: MARKER_NULL, MARKER_VLINE, MARKER_HLINE, MARKER_CROSS, MARKER_BOX or MARKER_HVLINE.
x	Specifies the x-value of the marker position
y	Specifies the y-value of the marker position

rsize	Specifies the size of the cross hair marker (MARKER_CROSS) and the box marker (MARKER_BOX) in UWP device coordinates.
npostype	Specifies the if the position of the marker is specified in physical coordinates, normalized coordinates or UWP device coordinates. Use one of the position constants: DEV_POS, PHYS_POS, NORM_GRAPH_POS, NORM_PLOT_POS.

The marker constants signify:

MARKER_NULL	An invisible marker
MARKER_VLINE	The marker is a vertical line extending from the top of the plot area to the bottom, passing through the x-value of the marker position.
MARKER_HLINE	The marker is a horizontal line extending from the left of the plot area to the right, passing through the y-value of the marker position.
MARKER_HVLINE	The marker combines both MARKER_VLINE and MARKER_HLINE, marking the data point with horizontal and vertical lines.
MARKER_CROSS	The marker is a cross hair centered on the marker position. Set the size of the cross hair using UWP device coordinates, in the object constructor, or later using the **setMarkerSize** method.
MARKER_BOX	The marker is a box centered on the marker position. Set the size of the box using UWP device coordinates, in the object constructor, or later using the **setMarkerSize** method.

Drop a marker anywhere on a plot by specifying the coordinates. The example below places a 10 pixel wide marker in the center of the plot area.

Simple marker example

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
CartesianCoordinates pTransform1 =
       new CartesianCoordinates( 0.0, 0.0, 10.0, 20.0);
pTransform1.AutoScale(Dataset1,  ChartObj.AUTOAXES_FAR, ChartObj.AUTOAXES_FAR);
.
.
.
double xpos = 5.0;
double ypos = 10.0;
Marker amarker = new Marker(pTransform1, ChartObj.MARKER_BOX, xpos, ypos,
                  10, ChartObj.PHYS_POS);
chartVu.AddChartObject(amarker);
```

Data Cursors

Class DataCursor

GraphObj

|

+--Marker

|

+-- DataCursor

Data cursors are an extension of the marker class. Data cursors combine the .Net mouse event delegates with the **Marker** class, creating a marker that tracks the mouse and updates dynamically. This constructor creates a new **DataCursor** object using the specified coordinate system, marker type and marker size.

DataCursor constructor

```
[C#]
public DataCursor(
   ChartView component,
   PhysicalCoordinates transform,
   int nmarkertype,
   double rsize
);
```

component	A reference to the **ChartView** object that the chart is placed in.
transform	The **PhysicalCoordinates** object associated with the data cursor.
nmarkdertype	The marker type. Use one of the **Marker** marker type constants: MARKER_VLINE .. MARKER_ BOX.
rsize	The size in UWP device coordinates of the MARKER_BOX and MARKER_CROSS style cursors.

See the **Marker** constructor description for more information about the **Marker** type constants.

Create the **DataCursor** object and then install it using the **ChartView.SetCurrentMouseListener** method. This adds the **DataCursor** object as a **MouseListener** to the **ChartView** object. Enable/Disable the function using **DataCursor.SetEnable** method. Call **DataCursor.SetCurrentMouseListener**(null) to remove the object as a mouse listener for the chart view.

Since the **DataCursor** class implements mouse event delegates, it has methods implementing the mouse events MouseMove, OnDoubleClick, OnClick, OnMouseDown, and OnMouseUp. The default usage of the **DataCursor** class creates the data marker when the mouse is pressed. As long as the mouse is pressed, the data cursor tracks the mouse position. Release the mouse button and the marker disappears. When using data markers it is often desirable to do additional things during the mouse events. In this case, you derive a new class from **DataCursor**, override the mouse events that you want to intercept, and add your own code to these events. Make sure you call the parents version of the same mouse event function so that the data cursor continues to track the mouse.

Simple data cursor example

[C#]

```
public class CustomChartDataCursor extends DataCursor {
// Create your own custom constructor and call the parent constructor

public CustomChartDataCursor(ChartView achartview,
             CartesianCoordinates thetransform,
             int nmarkertype,
              double rsize): base(achartview, thetransform, nmarkertype, rsize)
  {

  }

 // The mouse Released event should look like this
  public void OnMouseUp (PointerRoutedEventArgs event)
  {
     base.MouseReleased(mouseevent);
     // Add your own code here
  }
}
.
.
.

CustomChartDataCursor dataCursorObj =
    new CustomChartDataCursor( chartVu, pTransform1, ChartObj.MARKER_HVLINE, 8.0);
dataCursorObj.SetEnable(true);
```

```
chartVu.SetCurrentMouseListener(dataCursorObj);
```

A marker can be placed at any xy coordinate location in a graph. It is often desirable to place a marker at the exact location of a data point in one of the datasets plotted in the graph. Many applications require the user to click on the approximate location of a point, and then the software must find the data point nearest that click and mark it. The **DataCursor** and **Marker** classes, in combination with the plot objects CalcNearestPoint methods, accomplish this. The **DataCursor** class positions the mouse cursor and retrieves the initial xy coordinates. The CalcNearestPoint method for each plot object (**SimpleLinePlot**, **ScatterPlot**, etc.) in the graph determines the nearest data point to the mouse cursor for that object. Once all the plot objects are checked the data point nearest the mouse cursor position is marked by placing a **Marker** object at that exact xy location.

The example below extends the previous **Marker** and **DataCursor** examples. In this example, the OnMouseUp event of the subclassed **DataCursor** object processes the plot objects looking for the nearest point, and then places a **Marker** object and a numeric label at that point.

Marking a data point (Adapted from the MouseListeners example program (classes CustomChartDataCursor and DataCursors classes)

[C#]

```
   class CustomChartDataCursor : DataCursor
    {
        NumericLabel pointLabel;
        ChartFont textCoordsFont = new ChartFont("Microsoft Sans Serif", 8,
FontStyle.Normal);
        double rNumericLabelCntr = 0.0;
        SimpleLinePlot thePlot1;
        SimpleLinePlot thePlot2;

        public CustomChartDataCursor(ChartView achartview,
            CartesianCoordinates thetransform,
            SimpleLinePlot plot1,
            SimpleLinePlot plot2,
            int nmarkertype,
            double rsize)
            : base(achartview, thetransform, nmarkertype, rsize)
        {
            thePlot1 = plot1;
            thePlot2 = plot2;
        }

        public override void OnMouseUp(PointerRoutedEventArgs mouseevent)
        {
            NearestPointData nearestPointObj1 = new NearestPointData();
            NearestPointData nearestPointObj2 = new NearestPointData();
            Point2D nearestPoint = new Point2D(0, 0);
            ChartView chartview = GetChartObjComponent();
```

```
            bool bfound1 = false;
            bool bfound2 = false;

            // CurrentMouseButton is reset to 0 in base class, so save it for use below
            int currentmousebutton = CurrentMouseButton;
            base.OnMouseUp(mouseevent);

            // Use CurrentMouseButton to see what mouse button was released
            if (currentmousebutton == GetButtonMask())
            {

                // Find nearest point for each line plot object
                Point2D location = GetLocation();
                bfound1 = thePlot1.CalcNearestPoint(location,
                    ChartObj.FNP_NORMDIST, nearestPointObj1);
                bfound2 = thePlot2.CalcNearestPoint(location,
                    ChartObj.FNP_NORMDIST, nearestPointObj2);

                if (bfound1 && bfound2)
                {
                    // choose the nearest point
                    if (nearestPointObj1.GetNearestPointMinDistance() <
nearestPointObj2.GetNearestPointMinDistance())
                        nearestPoint = nearestPointObj1.GetNearestPoint();
                    else
                        nearestPoint = nearestPointObj2.GetNearestPoint();

                    // create marker object at place it at the nearest point
                    Marker amarker = new
                        Marker(GetChartObjScale(), MARKER_BOX, nearestPoint.GetX(),
nearestPoint.GetY(), 10.0, PHYS_POS);
                    chartview.AddChartObject(amarker);
                    rNumericLabelCntr += 1.0;
                    // Add a numeric label the identifies the marker
                    pointLabel = new NumericLabel(GetChartObjScale(), textCoordsFont,
rNumericLabelCntr,
                        nearestPoint.GetX(), nearestPoint.GetY(), PHYS_POS,
DECIMALFORMAT, 0);
                    // Nudge text to the right and up so that it does not write over
marker
                    pointLabel.SetTextNudge(5, -5);
                    chartview.AddChartObject(pointLabel);
                    chartview.Draw();
                }
            }
        }
    }
```

Another common reason for locating a data point is to display information associated with that data point. A good example is stock market data. A typical stock market display is a one-month chart of daily closing values for one or more stocks. You want to be able to click on a point in the chart and have the open, high, low and closing value for that day displayed in a pop-up box. The example program FinancialExample.OHLCFinPlot demonstrates how this can be done in a custom tool tip.. In another related example, the program PieCharts.SimplePieChart shows how to trap a click on a specific pie slice and display additional data for that slice using a **ChartText** object.

14. Moving Chart Objects, Data Points and Coordinate Systems

MoveObj
MoveData
MoveCoordinates

Many of the subclasses of **GraphObj** are moveable using the mouse. This includes the axis, legend, text, image, shape classes. If you add the necessary support to your program, you can click and drag the object around in the chart. This may or not be desirable, since a user can ruin a carefully constructed chart by dragging objects around. It is just an option though, that you can add to the program.

It is also possible to select a single data point in a simple plot object (**SimpleLinePlot**, **SimpleBarPlot**, **SimpleLineMarkerPlot** and **SimpleScatterPlot**) and move it with a click and drag operation of the mouse. Again, it is an option that you can add to the program if you want.

Starting in Revision 2.0 we have added the capability of moving the coordinates system of a graph, using the **MoveCoordinates** class. The move operation is analogous to the way you can change the latitude and longitude of an internet map by clicking and dragging it.

Moving Chart Objects

Class MoveObj

MouseListener
|
+--MoveObj

The **MoveObj** mouse listener traps a mouse pressed event and then searches through all of the **GraphObj** derived objects in the view. A rectangle highlights the first object that meets the filter criteria and intersects the mouse cursor. Hold the mouse button down and the rectangle tracks the mouse. Release the mouse button and the position of the graph object updates to reflect the new physical coordinates of the bounding rectangle.

If no *objectfilter* parameter is specified, the default object filter is “GraphObj”.

MoveObject constructors

```
[C#]
public MoveObj(
```

```
   ChartView component,
   MouseButton buttonmask,
   string object1filter
);

public MoveObj(
   ChartView component,
   MouseButton buttonmask
);

public MoveObj(
   ChartView component

);
```

component	A reference to the **ChartView** object that the chart is placed in.
buttonmask	Specifies the mouse button that is trapped to invoke a move.
objectfilter	The fully qualified class name of the base class that is used to filter the desired class objects. The string "ChartText" causes the routine to move only objects derived from the **ChartText** class. If you want to move only specific objects of a given class, create a special subclass of that class. Then create your moveable objects using that subclass. Then specify your class name, i.e. **MyTextClass,** using the string “MyTextClass”.

Create the **MoveObj** object and then install it using the **ChartView.SetCurrentMouseListener** method. This adds the **MoveObj** object as a **MouseListener** to the **ChartView** object. Enable/Disable the function using **MoveObj.SetEnable** method. Call **MoveObj.SetCurrentMouseListener**(null) to remove the object as a mouse listener for the chart view.

Not all **GraphObj** derived object are moveable. Call the **GraphObj.GetMoveableType** method and check to see if it returns **ChartObj.OBJECT_MOVEABLE**. Alternatively, you can call the **MoveObj.IsMoveableObject** method, passing in a reference to the object.

Most moveable objects move unrestricted in the x- and y direction. There are exceptions though. Axis objects move in the direction parallel to their current position, effectively changing the axis intercept, but not the extents of the axis endpoints. Axis labels always track their reference axis. The base axis defines the position of an **AxisTitle** text object and the chart view defines the position of a **ChartTitle** text object. Attemp to move these objects and they revert to their original centered positions. If you require moveable chart and axis titles, use the generic **ChartText** class instead of the title classes.

Moving objects example (Adapted from the MouseListeners.MoveObjects class)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
CartesianCoordinates pTransform1 =
       new CartesianCoordinates( 0.0, 0.0, 10.0, 20.0);
pTransform1.AutoScale(Dataset1,  ChartObj.AUTOAXES_FAR,
                      ChartObj.AUTOAXES_FAR);
.
.
.
MoveObj mouselistener = new MoveObj(chartVu );
mouselistener.SetEnable(true);
mouselistener.SetMoveObjectFilter("GraphObj");
mouselistener.ChartObjAttributes = new ChartAttribute(Colors.Black);
chartVu.SetCurrentMouseListener(mouselistener);
```

Moving Simple Plot Object Data Points

Class MoveData

MouseListener

|

+--MoveData

The **MoveData** mouse listener traps a mouse pressed event and searches through all of the data points of the plot objects in the view. The data point closest to the mouse cursor location is compared against a threshold value, 10 device units, or pixels, by default. If the data point is within the threshold, and as long as the moue buttons is held down, it tracks the mouse. Release the mouse button and the data value associated with the selected data point updates to reflect the new physical coordinates of the data point, and the plot is redrawn. Since the algorithm searches through every data point of every plot object in the view, do not expect it to work particularly fast with millions or even thousands of data points. The practical number of data points that can be searched is obviously dependent on the speed of the host computer.

MoveData constructors

```
[C#]
public MoveData(
   ChartView component,
   PhysicalCoordinates transform,
```

```
   MouseButton buttonmask
);

public MoveData(
   ChartView component,
   PhysicalCoordinates transform
);
```

component	A reference to the **ChartView** object that the chart is placed in.
transform	The **PhysicalCoordinates** object associated with the **MoveData** object.
buttonmask	Specifies the mouse button that is trapped to invoke a move.

Create the **MoveData** object and then install it using the **ChartView.SetCurrentMouseListener** method. This adds the **MoveData** object as a **MouseListener** to the **ChartView** object. Enable/Disable the function using **MoveData.SetEnable** method. Call **MoveData.SetCurrentMouseListener**(null) to remove the object as a mouse listener for the chart view. Set the threshold distance for deciding if the nearest data point found is a "hit" using the **MoveData.SetHitTestThreshold** method.

Moving datapoints example (See the class MoveDatapoint for a more complicated example)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
CartesianCoordinates pTransform1 =
       new CartesianCoordinates( 0.0, 0.0, 10.0, 20.0);
pTransform1.AutoScale(Dataset1,  ChartObj.AUTOAXES_FAR,
                     ChartObj.AUTOAXES_FAR);
.
.
MoveData mouselistener = new MoveData (chartVu, pTransform1 );
mouselistener.SetMarkerType( ChartObj.MARKER_CROSS);
mouselistener.SetMarkerSize(12);
mouselistener.SetMoveMode(ChartObj.MOVE_Y);
mouselistener.SetEnable(true);
chartVu.SetCurrentMouseListener(mouselistener);
```

Moving the Chart Coordinate System

Class MoveCoordinates

MouseListener
|
+--DataCursor
|
+--MoveCoordinates

The **MoveCoordinates** mouse listener traps a mouse pressed event. While the mouse is button is held down, the underlying coordinate system will track the move movements. Release the mouse button and the chart redraws one final time using the extents of the final coordinate system.

MoveCoordinates constructors

```
C#
public MoveCoordinates(
    ChartView component,
    PhysicalCoordinates transform,
    MouseButton buttonmask
)

public MoveCoordinates(
    ChartView component,
    PhysicalCoordinates transform
)
```

component	A reference to the **ChartView** object that the chart is placed in.
transform	The coordinate system underlying the chart.
buttonmask	Specifies the mouse button that is trapped to invoke a move.

Create the **MoveCoordinates** object and then install it using the **ChartView.SetCurrentMouseListener** method. This adds the **MoveCoordinates** object as a **MouseListener** to the **ChartView** object. Enable/Disable the function using MoveCoordinates**.SetEnable** method. Call MoveCoordinates**.SetCurrentMouseListener**(null) to remove the object as a mouse listener for the chart view.

You can restrict the movement of the coordinate system to the x-dimension (`MOVE_X`), or the y-dimension (`MOVE_Y`), using the **MoveMode** property. Set **MoveMode** to MOME_XY for unrestricted movement.

If you have multiple coordinate systems in the chart, you can move them all simultaneously by setting the **MultiTransformMove** property true. Otherwise, only the coordinate system passed into the constructor is updated with the move information.

Moving the coordinate system (Extracted from the ZoomExamples.MoveCoordinates example)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
TimeSimpleDataset Dataset1 = new TimeSimpleDataset("First", tradingDay, stockPrice);

TimeCoordinates pTransform1 = new TimeCoordinates();
pTransform1.AutoScale(Dataset1, ChartObj.AUTOAXES_NEAR, ChartObj.AUTOAXES_FAR);.
.
.
.
MoveCoordinates movecoords = new MoveCoordinates(chartVu, pTransform1);
movecoords.SetEnable(true);
chartVu.SetCurrentMouseListener(movecoords);
```

15. Zooming and Magnification

ChartZoom
MagniView

Zooming is the interactive re-scaling of a charts physical coordinate system and the related axes based on limits defined by clicking and dragging a mouse inside the current graph window. A typical use of zooming is in applications where the initial chart displays a large number of data points. The user interacts with the chart, defining smaller and smaller zoom rectangles, zeroing in on the region of interest. The final chart displays axis limits that have a very small range compared to the range of the original, un-zoomed, chart.

Important zoom features include:

- Automatic recalculation of axis properties for tick mark spacing and axis labels..
- Zooming of time coordinates with smooth transitions between major scale changes: years->months->weeks->days->hours->minutes->seconds.
- Zooming of time coordinates that use a 5-day week and a non-24 hour day.
- Simultaneous zooming of an unlimited number of x- and y-coordinate systems and axes (super zooming).
- The user can recover previous zoom levels using a zoom stack.
- The zoomed coordinate system can be forced to maintain a fixed aspect ratio.
- User-defineable zoom limits prevent numeric under and overflows

Magnification is related to zooming because it is also the interactive re-scaling of a charts physical coordinate system. In this case, a fixed sized view window is passed over a source chart, analogous to passing a magnifying class over a map. The area under the view window is "magnified" and redisplayed in a separate target chart. The source chart does not re-scale, as in the case of zooming. Only the target chart gets rescaled, in response to the position of the view window position over the source chart.

Simple Zooming of a single physical coordinate system

Class ChartZoom

MouseListener
|
+--ChartZoom

The **ChartZoom** class implements .Net delegates for mouse events. It implements and uses the mouse events: MouseMove, MouseDown, and MouseUp. The default operation of the **ChartZoom** class starts the zoom operation on the MouseDown event; it draws the zoom rectangle MouseMove event; and terminates the zoom operation on the mouse released event. During the mouse released event, the zoom rectangle is converted from device units into the chart physical coordinates and this information is stored and optionally used to rescale the chart scale and all axis objects that reference the chart scale. If four axis objects reference a single chart scale, for example when axes bound a chart on all for sides, all four axes re-scale to match the new chart scale.

Special Note

In other versions of this software, the zoom rectangle is drawn using the computers XOR mode. An XOR drawing mode is not supported in the UWP graphics libraries, so we no longer use it. The zoom rectangle is drawn using normal drawing techniques, which means that the entire chart must be redrawn many times during the click and drag of the zoom rectangle. This is much slower than the original XOR technique, where the chart did not have to be redrawn.

ChartZoom constructor

The constructor below creates a zoom object for a single chart coordinate system.

```
[C#]
public ChartZoom(
   ChartView component,
   PhysicalCoordinates transform,
   bool brescale
);
```

component	A reference to the **ChartView** object that the chart is placed in.
transform	The **PhysicalCoordinates** object associated with the scale being zoomed.
brescale	True designates that the scale should be re-scaled, once the final zoom rectangle is ascertained.

Enable the zoom object after creation using the **ChartZoom.SetEnable(true)** method.

Retrieve the physical coordinates of the zoom rectangle using the **ChartZoom** GetZoomMin and GetZoomMax methods. Restrict zooming in the x- or y-direction using the SetZoomXEnable and SetZoomYEnable methods. Set the rounding mode associated with rescale operations using the SetZoomXRoundMode and SetZoomYRoundMode methods. Call the **ChartZoom.PopZoomStack** method at any time and the chart scale reverts to the minimum and maximum values of the previous zoom operation. Repeated calls to the PopZoomStack method return the chart scale is to its original condition, after which the PopZoomStack method has no effect.

Integrated zoom stack processing
Starting with Revision 2.0, zoom stack processing is internal to **ChartZoom** class. There is no need to subclass the **ChartZoom** class in order to implement a zoom stack. Just set the ChartZoom.InternalZoomStackProcessing property true.

```
zoomObj.InternalZoomStackProcesssing = true;
```

Return to a previous zoom level by right clicking the mouse. Change the zoom stack button using the ZoomStackButtonMask property. Setting it to MouseButton.Left, MouseButton.Right or MouseButton.Middle.

Aspect Ratio Correction
Starting with Revision 2.0, you can force the zoom rectangle to maintain a fixed aspect ratio. Use the ChartZoom.ArCorrectionMode property to specify the aspect ratio correction mode.

ZOOM_NO_AR_CORRECTION	Allow the x- and y-dimension of the zoom rectangle to change the overall charts physical aspect ratio. This is the default mode, and the only mode supported prior to Revision 2.0.
ZOOM_X_AR_CORRECTION	Track the x-dimension of the zoom rectangle and calculate the y-dimension in order to maintain a fixed aspect ratio.
ZOOM_Y_AR_CORRECTION	Track the y-dimension of the zoom rectangle and calculate the x-dimension in order to maintain a fixed aspect ratio.

The target aspect ratio is the aspect ratio of the coordinate system(s) at the time the **ChartZoom** object is intitialized.

```
zoomObj.ArCorrectionMode = ChartObj.ZOOM_X_AR_CORRECTION
```

Simple zoom example (Adapted from the SimpleZoom example)
In this example, a new class derives from the **ChartZoom** class and the MousePressed event overridden. The event invokes the PopZoomStack method. Otherwise, the default operation of the **ChartZoom** class controls everything else.

[C#]

```
.
.
.
ChartZoom zoomObj = new ChartZoom (chartVu, pTransform1, true);
zoomObj.SetButtonMask(MouseButton.Left);
zoomObj.SetZoomYEnable(true);
zoomObj.SetZoomXEnable(true);
zoomObj.SetZoomXRoundMode(ChartObj.AUTOAXES_FAR);
zoomObj.SetZoomYRoundMode(ChartObj.AUTOAXES_FAR);
```

```
zoomObj.SetEnable(true);
zoomObj.SetZoomStackEnable(true);
// set range limits to 1000 ms, 1 degree
zoomObj.SetZoomRangeLimitsRatio(new Dimension(1.0, 1.0));
zoomObj.InternalZoomStackProcesssing = true;
chartVu.SetCurrentMouseListener(zoomObj);
```

Super Zooming of multiple physical coordinate systems

The **ChartZoom** class also supports the zooming of multiple physical coordinate systems (*super zooming*). During the mouse released event, the zoom rectangle is converted from device units into the physical coordinates of each scale, and this information is used to re-scale each coordinate system, and the axis objects associated with them.

Use the constructor below in order to super zoom a chart that has multiple coordinate systems and axes.

ChartZoom constructor

```
[C#]
public ChartZoom(
   ChartView component,
   PhysicalCoordinates[] transforms,
   bool brescale
);
```

component	A reference to the **ChartView** object that the chart is placed in.
transforms	An array, size numtransforms, of the **PhysicalCoordinates** objects associated with the zoom operation.
brescale	True designates that the all of the scales should be re-scaled, once the final zoom rectangle is ascertained.

Call the **ChartZoom.SetEnable(true)** method to enable the zoom object.

Restrict zooming in the x- or y-direction using the SetZoomXEnable and SetZoomYEnable methods. Set the rounding mode associated with rescale operations using the SetZoomXRoundMode and SetZoomYRoundMode methods. Call the **ChartZoom.PopZoomStack** method at any time and the chart scale reverts to the minimum and maximum values of the previous zoom operation. Repeated calls to the PopZoomStack method return the chart scale is to its original condition, after which the PopZoomStack method has no effect.

Starting with Revision 2.0, zoom stack processing is internal to **ChartZoom** class. There is no need to subclass the **ChartZoom** class in order to implement a zoom stack. Just set the ChartZoom.InternalZoomStackProcessing property true.

```
zoomObj.InternalZoomStackProcesssing = true;
```

Return to a previous zoom level by right clicking the mouse. Change the zoom stack button using the ZoomStackButtonMask property. Setting it to MouseButton.Left, MouseButton.Right or MouseButton.Middle.

Super zoom example (Adapted from the SuperZoom example)
In this example, a new class derives from the **ChartZoom** class and the MousePressed event overridden. The event invokes the PopZoomStack method. Otherwise, the default operation of the **ChartZoom** class controls everything else.

[C#]

```
public class SuperZoom
{
.
.
SimpleDataset Dataset1 = new SimpleDataset("First", x1,y1);
SimpleDataset Dataset2 = new SimpleDataset("Second",x1,y2);
SimpleDataset Dataset3 = new SimpleDataset("Third", x1,y3);
SimpleDataset Dataset4 = new SimpleDataset("Fourth",x1,y4);
SimpleDataset Dataset5 = new SimpleDataset("Fifth", x1,y5);

CartesianCoordinates pTransform1 =
       new CartesianCoordinates( ChartObj.LINEAR_SCALE, ChartObj.LINEAR_SCALE);
pTransform1.AutoScale(Dataset1, ChartObj.AUTOAXES_FAR, ChartObj.AUTOAXES_FAR);

CartesianCoordinates pTransform2 =
       new CartesianCoordinates( ChartObj.LINEAR_SCALE, ChartObj.LINEAR_SCALE);
pTransform2.AutoScale(Dataset2, ChartObj.AUTOAXES_FAR, ChartObj.AUTOAXES_FAR);
CartesianCoordinates pTransform3 =
       new CartesianCoordinates( ChartObj.LINEAR_SCALE, ChartObj.LINEAR_SCALE);
pTransform3.AutoScale(Dataset3, ChartObj.AUTOAXES_FAR, ChartObj.AUTOAXES_FAR);
CartesianCoordinates pTransform4 =
       new CartesianCoordinates( ChartObj.LINEAR_SCALE, ChartObj.LINEAR_SCALE);
pTransform4.AutoScale(Dataset4, ChartObj.AUTOAXES_FAR, ChartObj.AUTOAXES_FAR);
CartesianCoordinates pTransform5 =
       new CartesianCoordinates( ChartObj.LINEAR_SCALE, ChartObj.LINEAR_SCALE);
pTransform5.AutoScale(Dataset5, ChartObj.AUTOAXES_FAR, ChartObj.AUTOAXES_FAR);
.
.

CartesianCoordinates []transformArray =
       {pTransform1, pTransform2,pTransform3,pTransform4,pTransform5};

ChartZoom zoomObj = new ChartZoom (chartVu, transformArray, true);
zoomObj.SetButtonMask(MouseButton.Left);
zoomObj.SetZoomYEnable(true);
zoomObj.SetZoomXEnable(true);
zoomObj.SetZoomXRoundMode(ChartObj.AUTOAXES_FAR);
zoomObj.SetZoomYRoundMode(ChartObj.AUTOAXES_FAR);
zoomObj.SetEnable(true);
zoomObj.SetZoomStackEnable(true);
zoomObj.InternalZoomStackProcesssing = true;
chartVu.SetCurrentMouseListener(zoomObj);
```

.
.
.

Limiting the Zoom Range

A zoom window needs to have zoom limits placed on the minimum allowable zoom range for the x- and y-coordinates. Unrestricted, or infinte zooming can result in numeric under and overflows. The default minimum allowable range resuling from a zoom operation is 1/1000 of the original coordinate range. Change this value using the **ChartZoom.SetZoomRangeLimitsRatio** method. The minimum allowable range for this value is approximately 1.0e-9. Another way to set the minimum allowable range is to specify explicit values for the x- and y-range using the **ChartZoom.SetZoomRangeLimits** method. Specify the minimum allowable zoom range for a time axis in milliseconds, for example **ChartZoom.SetZoomRangeLimits(new Dimension(1000, 0.01))** sets the minimum zoom range for the time axis to 1 second and for the y-axis to 0.01. The utility method **ChartCalendar.GetCalendarWidthValue** is useful for calculating the milliseconds for any time base and any number of units. The code below sets a minimum zoom range of 45 minutes.

[C#]

```
double minZoomTimeRange  =
         ChartCalendar.GetCalendarWidthValue(ChartObj.MINUTE, 45);
double minZoomYRange = 0.01;
Dimension zoomLimits = new Dimension(minZoomTimeRange, minZoomYRange);
zoomObj.SetZoomRangeLimits(zoomLimits);
```

Magnifying a portion of a chart in a separate window

Class MagniView

MouseListener
|
+--MagniView

The **MagniView** class needs two chart areas in order to work: a source chart area, which contains the full scale display of the chart, and the target chart area, which will contain a magnified view of the chart.

The **MagniView** class starts the magnify operation on the OnMouseDown event, positioning the lower left corner of a magnify rectangle on top of the source chart. Immediately, a magnified view of the chart will appear in the target chart. As the mouse moves the magnify rectangle, the target chart constantly updates to reflect the current view window. Once the mouse button is released, the target chart remains at its current values.

MagniView constructor

The constructor below creates a **MagniView** object for a single chart coordinate system.

```
C#
public MagniView(
    ChartView source,
    ChartView target,
    PhysicalCoordinates transform,
    Dimension magnirect
)
```

source	A reference to the source **ChartView** object that the chart is placed in.
target	A reference to the target **ChartView** object that the magnified view of the chart is placed in.
transform	The source **PhysicalCoordinates** object associated with the scale being magnified.
magnirect	The rectangle, in physical coordinates, of the magnify cursor.

Enable the magnify object after creation using the **MagniView.SetEnable(true)** method.

Retrieve the physical coordinates of the magnify rectangle using the **MagniView** GetMagniMin and GetMagniMax methods. Restrict magnification in the x- or y-direction using the SetMagniXEnable and SetMagniYEnable methods. Set the rounding mode associated with rescale operations using the SetMagniXRoundMode and SetMagniYRoundMode methods.

In order to use the **MagniView** class, you need two instances of the underlying chart. Place one above, or next to the other, on a parent window, as done in the example. The first instance of the chart should be considered the source chart, and the second instance of the chart should be considered the target. This way you end up with two charts with an identical set of axes and plot ojects.

Simple magnify example, file ZoomExamples.MainPage.xaml.cs (Adapted from the ZoomExamples example)

[C#]

```
MagniViewChart mv1 = null;
MagniViewChart mv2 = null;

// In this example we make the common to both charts, because we must use the same data
for both
```

```
// instances of this chart. You will probably be initializing the data externally.
Regardless,
// of how you do it, when using the MagniView class you will need two instances of the
same
 //  class with the same data.

 private double[] x1 = null;
 private double[] y1 = null;

 private double[] y2 = null;
 private SimpleDataset Dataset1;

 private SimpleDataset Dataset2;
.
.
.

public void InitializeMagniViewCharts()
{
  mv1 = new MagniViewChart(magniView1);
  mv2 = new MagniViewChart(magniView2);

  InitializeData();

  // Define charts using data
  mv1.InitializeChart(Dataset1, Dataset2, "Click and drag on the top graph using left
mouse button.", "", false);
  mv2.InitializeChart(Dataset1, Dataset2, "The area bounded by the mouse 'magnify' icon
is displayed full-scale in the bottom graph.", "", true);

  // Hook-up the MagniView class to the source and target classes
  MagniView magnifyObj = new MagniView(magniView1, magniView2, mv1.pTransform1, new
Dimension(0.05, 0.2));
  ChartAttribute attrib1 = new ChartAttribute(Colors.Black, 1, ChartObj.LS_SOLID);
  magnifyObj.ChartObjAttributes = attrib1;
  magnifyObj.SetButtonMask(MouseButton.Left);
  magnifyObj.SetEnable(true);
  magnifyObj.UpdateDuringDrag = true;
  magniView1.SetCurrentMouseListener(magnifyObj);

  magniView1.PreferredSize = new Size(1000, 300);
  magniView2.PreferredSize = new Size(1000, 300);
}
```

16. Data Tooltips

DataToolTip

Tooltip is a catchall phrase for a popup window that displays useful information about an object. A data tooltip is a popup box that displays the value of a data point in a chart. The data value can consist of the x-value, the y-value, or both values for a given point in a chart. The tooltip values are displayed using the numeric and time formats supported by the **NumericLabel** and **TimeLabel** classes.

Simple Data Tooltips

Class DataToolTip

MouseListener
|
+-**DataToolTip**

The **DataToolTip** class implements .Net mouse event delegates. The ChartView class hooks into the its drawing surface MouseDown, MouseUp and MouseMove events. The default operation of the **DataToolTip** class traps the mouse pressed event. It calculates which chart object intersects the mouse cursor and which data points for the intersecting object are closest. Next, it pops up a window and displays the x-value and/or y-value representing the data point. When the mouse button is released, the MouseUp event, the tooltip popup window is deleted.

The tooltip data point search algorithm is complicated by the situation that many chart objects occupy a much larger area than the data point that is represented. Bars are a good example. A bar can occupy a large area, yet the actual data value represented by the bar is only a small point at the top. The tooltip algorithm searches for an intersection of the bar and the mouse cursor, not an intersection of the data point and the mouse cursor. If a hit on the bar is detected, the data values represented by the bar are used as the values displayed in the tooltip. The tooltip symbol will highlight the actual data point at the top of the bar. The tooltip window will always popup at the cursor location, not the data point location. If you click anywhere on a bar, the tooltip window will popup at that location and display the data value represented by the top of the bar.

The tooltip data point search algorithm works with both simple and group data. When used with simple plot objects (**SimpleLinePlot**, **SimpleBarPlot**, etc.) it locates the xy data point associated with the mouse event. When used with group plot objects it locates the x-value and the y-group value associated with the mouse event. It is able to differentiate between “stacked” group plot objects (**StackedBarPlot**, **StackedLinePlot**) and the other group plot objects that are not stacked

(**GroupBarPlot**, **MultiLinePlot**, **OHLCPlot**, **CandlestickPlot**, etc.). The tooltip values displayed in the tooltip window reflect the actual data values stored in the associated dataset and do not reflect the implicit summation that goes on in the display of stacked plot objects. You should not use symbols to highlight the tooltip data point for stacked objects since the position of the tooltip symbol in the chart will not take into account the stacked object summation.

DataToolTip constructors

The constructors below create a **DataToolTip** object.

```
[C#]
public DataToolTip(
   ChartView component
);

public DataToolTip(
   ChartView component,
   MouseButton buttonmask
);
```

component	A reference to the **ChartView** object that the chart is placed in.
buttonmask	Specifies the mouse button that is trapped to invoke a move.

Create the **DataToolTip** object and then install it using the **ChartView.SetCurrentMouseListener** method. This adds the **DataToolTip** object as a **MouseListener** to the **ChartView** object. Enable/Disable the function using **DataToolTip.SetEnable** method. Call **ChartView.SetCurrentMouseListener**(null) to remove the object as a mouse listener for the chart view.

Set the threshold distance for deciding if the nearest data point found is a "hit" using the **DataToolTip.SetHitTestThreshold** method.

The default values for the **DataToolTip** class assume the following:

- The left mouse button pressed event is the trigger for the tooltip. Change this using the DataToolTip.SetButtonMask method.
- The numeric format for the x- and y-values are controlled by separate NumericLabel class templates that are both initially set to the ChartObj.DECIMALFORMAT format with a decimal precision of 1. Change this by creating a NumericLabel or TimeLabel object that specifies how you want the number formatted. Set the x- and y-value templates independently using the DataToolTip.SetXValueTemplate and DataToolTip.SetYValue template methods.

- The tooltip will display just the y-value of the selected object. Change this to display the x-value or both the x and y-values using the DataToolTip.SetDataToolTipFormat method, specifying one of the data tooltip format constants: DATA_TOOLTIP_CUSTOM, DATA_TOOLTIP_X, DATA_TOOLTIP_Y, DATA_TOOLTIP_XY_ONELINE, DATA_TOOLTIP_TWOLINE, DATA_TOOLTIP_GROUP_MULTILINE, DATA_TOOLTIP_OHLC.
- The tooltip popup window uses a default Sans Serif font with a size of 12. The text is justified above and to the right of the mouse cursor. The default background color of the data tooltip is a pale yellow, RGB (255,255,204). Change this by replacing the ChartText object used as the template for the tooltip popup window. Use the DataToolTip.SetTextTemplate method to replace the default template with your own.
- The selected data point is highlighted using a ChartSymbol object, set to a default shape of ChartObj.SQUARE, a size of 8 and a color of black. Change this by replacing the ChartSymbol used as the template with one of your own.

Simple data tooltip example (Adapted from the MultiAxes example)
In this example, the tooltip will display the decimal y-value of the nearest data point when the left mouse button is pressed.

[C#]

```
ChartView chartVu;

DataToolTip datatooltip = new DataToolTip(chartVu);
chartVu.SetCurrentMouseListener(datatooltip);
```

Medium complex data tooltip example (Adapted from the LineFill example)
In this example, the tooltip will display the x-value of the data point as a date, and the y-value as currency. The x- and y-values are displayed on two separate lines, one above the other.

[C#]

```
ChartFont toolTipFont = new ChartFont("Microsoft Sans Serif", 10, FontStyle.Normal);
DataToolTip datatooltip = new DataToolTip(chartVu);
TimeLabel xValueTemplate = new TimeLabel( ChartObj.TIMEDATEFORMAT_MDY);
NumericLabel yValueTemplate = new NumericLabel( ChartObj.CURRENCYFORMAT,0);
datatooltip.GetToolTipSymbol().SetColor(Colors.Green);
datatooltip.SetXValueTemplate(xValueTemplate);
datatooltip.SetYValueTemplate(yValueTemplate);
datatooltip.SetDataToolTipFormat(ChartObj.DATA_TOOLTIP_XY_TWOLINE);
datatooltip.SetEnable(true);
chartVu.SetCurrentMouseListener(datatooltip);
```

Complex data tooltip example (Adapted from the OpeningScreen example)
In this example, the tooltip will display the x-value of the data point as a date, and the y-value as currency. The x- and y-values are displayed on one line, side by side.

[C#]

```
ChartFont toolTipFont = new ChartFont("Microsoft Sans Serif", 10, FontStyle.Normal);
DataToolTip datatooltip = new DataToolTip(chartVu);
TimeLabel xValueTemplate = new TimeLabel( ChartObj.TIMEDATEFORMAT_MDY);
NumericLabel yValueTemplate = new NumericLabel(ChartObj.CURRENCYFORMAT,2);
ChartText textTemplate = new ChartText(toolTipFont,"");
textTemplate.SetTextBgColor(Color.FromArgb(255,255,255, 204));
textTemplate.SetTextBgMode(true);
ChartSymbol toolTipSymbol =
       new ChartSymbol(null,  ChartObj.SQUARE, new ChartAttribute(Colors.Black));
toolTipSymbol.SetSymbolSize(5.0);
datatooltip.SetTextTemplate(textTemplate);
datatooltip.SetXValueTemplate(xValueTemplate);
datatooltip.SetYValueTemplate(yValueTemplate);
datatooltip.SetDataToolTipFormat(ChartObj.DATA_TOOLTIP_OHLC);
datatooltip.SetToolTipSymbol(toolTipSymbol);
datatooltip.SetEnable(true);
chartVu.SetCurrentMouseListener(datatooltip);
```

Custom Tooltip displays

It would be impossible to provide options for all possible tooltip displays. The **DataToolTip** class includes an option that enables the programmer to override the existing behavior of the class. The programmer is able to use the built in search routines to identify what plot object is selected, the dataset, the coordinate system and the actual data values associated with the plot object. Using this information the programmer can customize the text displayed in the **ChartText** object used to display the tooltip text.

Use the following steps to create a custom tooltip.

- Subclass the **DataToolTip** class with one of your own.
- Enable the custom mode by calling **DataToolTip.SetDataToolTipFormat**(ChartObj.DATA_TOOLTIP_CUSTOM) method.
- Override the **OnMouseDown** and **OnMouseUp** events and add the code needed to customize the display. In your custom **OnMouseDown** event, make sure you call **super**.**OnMouseDown** first, since this selects the plot object for you, and makes the plot objects coordinate system, dataset, and selected data point available using get methods. You can place your tooltip text in the **ChartText** object internal to the **DataToolTip** class. Get a reference to this object using the **DataToolTip.GetTextTemplate**() method. In your custom **OnMouseUp** event call **super.OnMouseUp** followed by a call to the **ChartView** repaint method (chartVu.UpdateDraw() in the example below);

Custom DataToolTip example (Adapted from the FinancialExamples.OHLCChart example)

In this example, a new class is derived from the **DataToolTip** class and the OnMouseDown and OnMouseUp events are overridden.

[C#]

```
ChartView chartVu;

       class CustomToolTip : DataToolTip
        {
            OHLCChart OHLCObj = null;

            public CustomToolTip(OHLCChart component)
                : base(component.chartVu)
            {

                OHLCObj = component;
            }

            public override void OnMouseUp(PointerRoutedEventArgs  mouseevent)
            {
                base.OnMouseUp(mouseevent);
                GetChartObjComponent().UpdateDraw();
            }

            public override void OnMouseDown(PointerRoutedEventArgs  mouseevent)
            {
                Point2D mousepos = new Point2D();

mousepos.SetLocation(mouseevent.GetCurrentPoint(OHLCObj.chartVu).Position.X,
                   mouseevent.GetCurrentPoint(OHLCObj.chartVu).Position.Y);

                base.OnMouseDown(mouseevent);

                ChartPlot selectedPlot = (ChartPlot)GetSelectedPlotObj();
                if (selectedPlot != null)
                {
                    int selectedindex = GetNearestPoint().GetNearestPointIndex();
                    PhysicalCoordinates transform = GetSelectedCoordinateSystem();
                    // Looking to the original arrays, because we just have the
selectedindex,
                    // yet we want to display stock O-H-L-C data, volume and NASDAQ. Only
one
                    // of these datasets can be selected at a time by the tooltip.
                    double open = OHLCObj.stockPriceData[0, selectedindex];
                    double high = OHLCObj.stockPriceData[1, selectedindex];
                    double low = OHLCObj.stockPriceData[2, selectedindex];
                    double close = OHLCObj.stockPriceData[3, selectedindex];
                    double nasdaq = OHLCObj.NASDAQData[selectedindex];
                    double volume = OHLCObj.stockVolumeData[selectedindex];
                    String openObj = ChartSupport.NumToString(open,
ChartObj.DECIMALFORMAT, 2, "");
                    String highObj = ChartSupport.NumToString(high,
ChartObj.DECIMALFORMAT, 2, "");
                    String lowObj = ChartSupport.NumToString(low, ChartObj.DECIMALFORMAT,
2, "");
                    String closeObj = ChartSupport.NumToString(close,
ChartObj.DECIMALFORMAT, 2, "");
```

```
                    String volumeObj = ChartSupport.NumToString(volume,
ChartObj.DECIMALFORMAT, 0, "");
                    String nasdaqObj = ChartSupport.NumToString(nasdaq,
ChartObj.DECIMALFORMAT, 2, "");
                    TimeLabel timelabel = new TimeLabel(transform,
OHLCObj.xValues[selectedindex], ChartObj.TIMEDATEFORMAT_STANDARD);

                    String tooltipstring = "Stock Data" + "\n";
                    tooltipstring += timelabel.GetTextString() + "\n";
                    tooltipstring += "Open " + openObj + "\n";
                    tooltipstring += "High " + highObj + "\n";
                    tooltipstring += "Low " + lowObj + "\n";
                    tooltipstring += "Close " + closeObj + "\n";
                    tooltipstring += "Volume " + volumeObj + "\n";
                    tooltipstring += "NASDAQ " + nasdaqObj;

                    CustomToolTipString = tooltipstring;
                    Draw();


                }
            }
        }

CustomToolTip stocktooltip =
        new CustomToolTip(chartVu, xValues, stockPriceData, NASDAQData,
                  stockVolumeData);
stocktooltip.SetDataToolTipFormat(ChartObj.DATA_TOOLTIP_CUSTOM);
stocktooltip.SetEnable(true);
chartVu.SetCurrentMouseListener(stocktooltip);
```

17. Pie and Ring Charts

PieChart
RingChart

Everyone is familiar with the ubiquitous pie chart. Pie charts are 1-dimensional, not because they are shallow, but because they represent a simple 1-dimensional series of numbers, {3, 5, 2, 7, 3, ...}, rather than the parametric set of data points { (3,2), (6,3), (7,3)…} used in the other plot types described in this software. The best use of pie charts involves data that has 10 or fewer elements. Otherwise, the text used to label the pie charts starts to overlap in adjacent, small pie wedges. The x-values of a dataset represent the datavalues for each pie wedge. The y-values of the dataset explode a pie wedge from its normal centered position.

A ring chart is a variant of the pie chart. Instead an entire circle (or pie) being the basis of the chart, a ring is used instead.

Using the Pie Chart Class

Class PieChart

GraphObj
|
+--ChartPlot
|
+--SimplePlot
|
+-- PieChart
|
+-- RingChart

The **PieChart** and **RingChart** classes extends the **ChartPlot** class and displays pie/ring charts. The x-values of the simple dataset used for data storage specify the pie/ring wedge values. The y-values of the dataset specify the "explode" percentage for each pie/ring wedge.

PieChart Constructor

```
[C#]
public PieChart(
   PhysicalCoordinates transform,
   SimpleDataset dataset,
   string[] spiestring1s,
   ChartAttribute[] attribs,
   int labelinout1,
   int pielabelformat
);
```

RingChart Constructor

```
[C#]
public RingChart(
   PhysicalCoordinates transform,
   SimpleDataset dataset,
   string[] spiestrings,
   ChartAttribute[] attribs,
   int labelinout1,
   int pielabelformat
);
```

transform The pie/ring chart is placed in the coordinate system defined by transform.

dataset The pie/ring chart represents the values in this dataset. The x-values of the simple dataset used for data storage specify the pie/ring wedge values. The y-values of the dataset specify the "explode" percentage for each pie/ring wedge.

spiestrings An array of strings, size dataset.GetNumberDatapoints(), used as labels for the pie/ring slices.

attribs An array of **ChartAttribute** objects, size dataset.GetNumberDatapoints() that specify the attributes (outline color and fill color) for each wedge of a pie/ring chart.

labelinout An array of integer, size dataset.GetNumberDatapoints(), specifying if a specific pie/ring slice text label is drawn inside the pie/ring slice, or outside of the pie/ring slice. Use one of the constants: PIELABEL_OUTSLICE or PIELABEL_INSLICE.

pielabelformat All pie/ring slice labels share the same format. Use one of the pie/ring slice label format constants:

PIELABEL_NONE Do not display and pie/ring slice text

PIELABEL_STRING Display only the pie/ring text strings, no numeric values

PIELABEL_NUMVALUE Display the pie/ring numeric value only, no pie text strings.

PIELABEL_STRINGNUMVAL Display the pie/ring text string and numeric value.

A pie/ring chart uses a default **CartesianCoordinates** object. Center it in the window using the **CartesianCoordinates.SetGraphBorderDiagonal** method. Format the text used to label the pie/ring chart, both the strings and the numeric values, using a **NumericLabel** template set using the **PieChart.SetPlotLabelTemplate** method. Change the starting position of the first pie/ring

wedge from the default value of 0.0 (3:00 position) using the **PieChart.SetStartPieSliceAngle** method. The **PieChart.CalcNearestPoint** method can find the pie/ring wedge nearest a specified point, usually the result of a mouse click. See the **PieCharts.SimplePieChart** example program.

Simple pie chart (extracted from the example program PieCharts, class SimplePieChart). A simlar example for a RingChart is found in the example program NewDemosRev2.SimpleRingChart.

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
int  numPoints = 5;
String []sPieStrings = {"Technology", "Retail", "Banking",
       "Automotive", "Aerospace"};
ChartAttribute []attribs = new ChartAttribute[5];
ChartFont theFont;
Color []colorArray = {Colors.Red, Colors.Blue, Colors.Cyan,
       Colors.Yellow, Colors.Green, Colors.DarkGray, Colors.LightGray,
       Colors.Magenta, Colors.Orange, Colors.Pink};
ChartText techLabel;
ChartText retailLabel;
ChartText bankLabel;
ChartText aeroLabel;
ChartText autoLabel;

theFont = new ChartFont("Microsoft Sans Serif", 10,  FontStyle.Normal, FontWeights.Bold);

double []x1 = new double[numPoints];
double []y1 = new double[numPoints];
int i;

for (i=0; i < numPoints; i++)
{
    attribs[i] = new ChartAttribute (Colors.Black, 1,0);
    attribs[i].SetFillColor(colorArray[i]);
}
x1[0] = 5.8; y1[0] = 0.2;
x1[1] = 2.2; y1[1] = 0.0;
x1[2] = 3.5; y1[2] = 0.0;
x1[3] = 4.2; y1[3] = 0.0;
x1[4] = 3.7; y1[4] = 0.0;
SimpleDataset Dataset1 = new SimpleDataset("First",x1,y1);
CartesianCoordinates pTransform1 = new CartesianCoordinates();
pTransform1.SetGraphBorderDiagonal(0.1, .2, .9, 0.9) ;

Background background1 = new Background( pTransform1,
       ChartObj.GRAPH_BACKGROUND,Color.FromArgb(255,0,120,70),
       Color.FromArgb(255,0,40,30), ChartObj.Y_AXIS);
chartVu.AddChartObject(background1);

PieChart thePlot1 =
       new PieChart(pTransform1, Dataset1, sPieStrings,attribs,
ChartObj.PIELABEL_OUTSLICE, ChartObj.PIELABEL_STRINGNUMVAL);
thePlot1.SetStartPieSliceAngle(-45);

NumericLabel labeltemplate = new NumericLabel();
labeltemplate.SetNumericFormat(ChartObj.CURRENCYFORMAT);
labeltemplate.SetDecimalPos(1);
labeltemplate.SetTextFont(theFont);
```

```
thePlot1.SetPlotLabelTemplate(labeltemplate);
thePlot1.SetLabelInOut(0,ChartObj.PIELABEL_INSLICE);
thePlot1.SetLabelInOut(1,ChartObj.PIELABEL_INSLICE);
thePlot1.SetLabelInOut(2,ChartObj.PIELABEL_INSLICE);
thePlot1.SetLabelInOut(3,ChartObj.PIELABEL_INSLICE);
thePlot1.SetLabelInOut(4,ChartObj.PIELABEL_INSLICE);
chartVu.AddChartObject(thePlot1);
```

18. Polar and Antenna Plots

PolarPlot
- **PolarLinePlot**
- **PolarScatterPlot**

AntennaPlot
- **AntennaLinePlot**
- **AntennaScatterPlot**
- **AntennaLineMarkerPlot**

Polar charts play an important in engineering applications involving electronics and advanced control systems. Polar charts give a visual interpretation to mathematical problems involving trigonometric functions and complex numbers. Antenna charts are used to display the operating characteristics of antennas.

The **PolarPlot** class is an abstract class representing plot types that use data organized as arrays of x- and y-values, where an x-value represents the magnitude of a point in polar coordinates, and the y-value represents the angle of a point in polar coordinates. Polar plots types include: line plots and scatter plots.

The polar angle values stored as the y-values in the **SimpleDataset** should be in radians. If the raw data is in degrees, convert the data to radians using the ChartSupport.**ToRadians** method.

The **AntennaPlot** class is an abstract class representing plot types that use data organized as arrays of x- and y-values, where an x-value represents the radial value of a point in antenna coordinates, and the y-value represents the angle of a point in antenna coordinates. Antenna plots types include: line plots, scatter plots, line marker plots, and annotations.

The antenna angle values stored as the y-values in the **SimpleDataset** should be specified in degrees. If the raw data is in radians, convert the data to degrees using the ChartSupport.ToDegrees method.

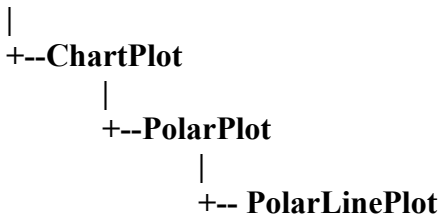
If you plan to create polar charts, you need to also familiarize yourself with the polar charting classes describe in the other chapters of this manual. These are the chapters on coordinate systems (**PolarCoordinates** in Chapter 4), axes (**PolarAxes** in Chapter 7), axis labels (**PolarAxesLabels** in Chapter 8) and grids (**PolarGrids** in Chapter 9). The same is true if you plan to create antenna charts. Refer to the documentation in the chapters on coordinate systems (**AntennaCoordinates** in Chapter 4) , axes (**AntennaAxes** in Chapter 7), axis labels (**AntennaAxesLabels** in Chapter 8) and grids (**AntennaGrids** in Chapter 9).

The **ChartZoom**, **MagniView,** and **MoveCoordinates** classes require a rectangular coordinate system and will not work with polar and antenna charts. The **MoveData** class does work with polar and antenna charts.

Polar Plots

Class PolarLinePlot

GraphObj



The **PolarLinePlot** class is a concrete implementation of the **PolarPlot** class and displays data in a simple line plot format. The lines drawn between adjacent data points use polar coordinate interpolation.

PolarLinePlot constructor

```
[C#]
public PolarLinePlot(
   PolarCoordinates transform,
   SimpleDataset dataset,
   ChartAttribute attrib
);
```

transform	The coordinate system for the new **PolarLinePlot** object.
dataset	The polar line plot represents the polar coordinate values in this dataset. The x-values of the dataset represent the magnitudes of the points and the y-values the polar angles in radians.
attrib	Specifies the attributes (line color and line style) for the line plot.

The polar line plot class interpolates between adjacent data points in polar coordinates and not using straight lines as in the Cartesian coordinate plotting functions. This gives the lines between adjacent data points in a polar plot a curved look.

Polar line plot and scatter plot chart (extracted from the example program PolarCharts. PolarLineAndScatterChart)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
int nump1 = 100;
double []mag1 = new double[nump1];
double []ang1 = new double[nump1];
int i;
for (i=0; i < nump1; i++)
{
    ang1[i] = ChartSupport.ToRadians((double)i * (360.0/ (double)nump1));
    mag1[i] = Math.Abs(30 * (Math.Sin(2*(ang1[i])) * Math.Cos(2 * (ang1[i]))));
}
theFont = new ChartFont("Microsoft Sans Serif", 10,  FontStyle.Normal, FontWeights.Bold);


SimpleDataset Dataset1 = new SimpleDataset("First",mag1,ang1);
PolarCoordinates pPolarTransform = new PolarCoordinates();

pPolarTransform.SetGraphBorderDiagonal(0.25, .20, .75, 0.8) ;

Background background =
    new Background( pPolarTransform, ChartObj.GRAPH_BACKGROUND,
    Colors.White);
chartVu.AddChartObject(background);

pPolarTransform.AutoScale(Dataset1);
PolarAxes pPolarAxis = pPolarTransform.GetCompatibleAxes();
chartVu.AddChartObject(pPolarAxis);

PolarGrid pPolarGrid = new PolarGrid (pPolarAxis, PolarGrid.GRID_MAJOR);
chartVu.AddChartObject(pPolarGrid);

PolarAxesLabels pPolarAxisLabels = (PolarAxesLabels) pPolarAxis.GetCompatibleAxesLabels();
chartVu.AddChartObject(pPolarAxisLabels);

ChartAttribute attrib1 = new ChartAttribute (Colors.Blue, 2,0);
PolarLinePlot thePlot1 = new PolarLinePlot(pPolarTransform, Dataset1, attrib1);
chartVu.AddChartObject(thePlot1);

ChartAttribute attrib2 = new ChartAttribute (Colors.Red, 1,0,Colors.Red);
attrib2.SetFillFlag(true);
PolarScatterPlot thePlot2 =
       new PolarScatterPlot(pPolarTransform,  Dataset1,ChartObj.CIRCLE,attrib2);
chartVu.AddChartObject(thePlot2);
```

Class PolarScatterPlot

GraphObj

|

+--ChartPlot

|

+--PolarPlot

|

+-- PolarScatterPlot

The **PolarScatterPlot** class is a concrete implementation of the **PolarPlot** class and displays data in a simple scatter plot format.

PolarScatterPlot constructor

```
[C#]
public PolarScatterPlot(
   PolarCoordinates transform,
   SimpleDataset dataset,
   int symtype,
   ChartAttribute attrib
);
```

transform	The coordinate system for the new **PolarScatterPlot** object.
dataset	The polar scatter plot represents the polar coordinate values in this dataset. The x-values of the dataset represent the magnitudes of the points and the y-values the polar angles in radians.
symtype	The symbol used in the scatter plot. Use one of the scatter plot symbol constants: NOSYMBOL, SQUARE, TRIANGLE, DIAMOND, CROSS, PLUS, STAR, LINE, HBAR, VBAR, CIRCLE.
attrib	Specifies the attributes (size, line and fill color) for the scatter plot.

See previous example for a programming example using PolarScatterPlot.

Antenna Plots

Class AntennaLinePlot

GraphObj
|
+--ChartPlot
|
+--AntennaPlot
|
+-- AntennaLinePlot

The **AntennaLinePlot** class is a concrete implementation of the **AntennaPlot** class and displays data in a simple line plot format. The lines drawn between adjacent data points use antenna coordinate interpolation.

AntennaLinePlot constructor

```
[C#]
public AntennaLinePlot(
   AntennaCoordinates transform,
   SimpleDataset dataset,
   ChartAttribute attrib
);
```

transform	The coordinate system for the new **AntennaLinePlot** object.
dataset	The antenna line plot represents the antenna coordinate values in this dataset. The x-values of the dataset represent the radial values and the y-values represent the antenna angular values in degrees.
attrib	Specifies the attributes (line color and line style) for the line plot.

The antenna line plot class interpolates between adjacent data points in antenna coordinates and not using straight lines as in the Cartesian coordinate plotting functions. This gives the lines between adjacent data points in a antenna plot a curved look.

Antenna line plot and scatter plot chart (extracted from the example program PolarCharts.AntennaLineMarkerChart)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
Dataset1 = new SimpleDataset("First", mag1, ang1);
Dataset2 = new SimpleDataset("Second", mag2, ang1);

SimpleDataset[] datasetarray = { Dataset1, Dataset2 };
pAntennaTransform = new AntennaCoordinates();
pAntennaTransform.AutoScale(datasetarray, ChartObj.AUTOAXES_FAR);

pAntennaTransform.SetGraphBorderDiagonal(0.25, .15, .75, 0.85);

Background background = new Background(pAntennaTransform, ChartObj.GRAPH_BACKGROUND,
Colors.White);
chartVu.AddChartObject(background);

AntennaAxes pAntennaAxis = pAntennaTransform.GetCompatibleAxes();
pAntennaAxis.LineColor = Colors.Black;
chartVu.AddChartObject(pAntennaAxis);

AntennaGrid pAntennaGrid = new AntennaGrid(pAntennaAxis, AntennaGrid.GRID_ALL);
pAntennaGrid.ChartObjAttributes = new ChartAttribute(Colors.LightBlue, 1,
ChartObj.LS_SOLID);
chartVu.AddChartObject(pAntennaGrid);

AntennaAxesLabels pAntennaAxisLabels =
(AntennaAxesLabels)pAntennaAxis.GetCompatibleAxesLabels();
chartVu.AddChartObject(pAntennaAxisLabels);

Color transparentRed = Color.FromArgb(255,180, 255, 0, 0);
Color transparentBlue = Color.FromArgb(255,180, 0, 0, 255);
ChartAttribute attrib1 = new ChartAttribute(transparentRed, 1, ChartObj.LS_SOLID);
attrib1.SymbolSize = 7;
```

```
ChartAttribute attrib2 = new ChartAttribute(Colors.Blue, 1, ChartObj.LS_SOLID,
Colors.Blue);
attrib2.SymbolSize = 7;

ChartAttribute attrib3 = new ChartAttribute(Colors.Yellow, 3, ChartObj.LS_SOLID,
Colors.Yellow);
ChartAttribute attrib4 = new ChartAttribute(Colors.MediumPurple, 2, ChartObj.LS_DOT_1_4,
Colors.MediumPurple);
AntennaLinePlot thePlot1 = new AntennaLinePlot(pAntennaTransform);
thePlot1.InitAntennaLinePlot(Dataset1, attrib1);
chartVu.AddChartObject(thePlot1);

AntennaScatterPlot thePlot2 = new AntennaScatterPlot(pAntennaTransform);
thePlot2.InitAntennaScatterPlot(Dataset1, ChartObj.SQUARE, attrib2);
chartVu.AddChartObject(thePlot2);
```

Class AntennaScatterPlot

GraphObj
|
+--ChartPlot
|
+--AntennaPlot
|
+-- AntennaScatterPlot

The **AntennaScatterPlot** class is a concrete implementation of the **AntennaPlot** class and displays data in a simple scatter plot format.

AntennaScatterPlot constructor

```
[C#]
public AntennaScatterPlot(
   AntennaCoordinates transform,
   SimpleDataset dataset,
   int symtype,
   ChartAttribute attrib
);
```

transform	The coordinate system for the new **AntennaScatterPlot** object.
dataset	The antenna scatter plot represents the antenna coordinate values in this dataset. The x-values of the dataset represent the radial values of the points and the y-values the angular values in degrees.
symtype	The symbol used in the scatter plot. Use one of the scatter plot symbol constants: NOSYMBOL, SQUARE, TRIANGLE, DIAMOND, CROSS, PLUS, STAR, LINE, HBAR, VBAR, CIRCLE.
attrib	Specifies the attributes (size, line and fill color) for the scatter plot.

See previous example for a programming example using AttennaScatterPlot.

Class AntennaLineMarkerPlot

GraphObj
|
+--ChartPlot
|
+--AntennaPlot
|
+-- AntennaLineMarkerPlot

The **AntennaLineMarkerPlot** class is a concrete implementation of the **AntennaPlot** class and displays data in a simple line marker plot format.

AntennaScatterPlot constructor

```
[C#]
public AntennaLineMarkerPlot(
   AntennaCoordinates transform,
   SimpleDataset dataset,
   int symtype,
   ChartAttribute lineattrib,
   ChartAttribute symbolattrib,

);
```

transform	The coordinate system for the new **AntennaLineMarkerPlot** object.
dataset	The line marker plot represents the values in this dataset.
symtype	The symbol used in the line marker plot. Use one of the scatter plot symbol constants: NOSYMBOL, SQUARE, TRIANGLE, DIAMOND, CROSS, PLUS, STAR, LINE, HBAR, VBAR, CIRCLE.
lineattrib	Specifies the attributes (line color and line style) for the line part of the line marker plot.
symbolattrib	Specifies the attributes (line and fill color) for the symbol part of the line marker plot.

Antenna line marker plot (extracted from the example program PolarCharts.AntennaLineMarkerChart)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
Color transparentRed = Color.FromArgb(180, 255, 0, 0);
Color transparentBlue = Color.FromArgb(180, 0, 0, 255);

ChartAttribute attrib1 = new ChartAttribute(transparentRed, 1, ChartObj.LS_SOLID);
attrib1.SymbolSize = 7;
ChartAttribute attrib2 = new ChartAttribute(Colors.Blue, 1, ChartObj.LS_SOLID,
Colors.Blue);
attrib2.SymbolSize = 7;

ChartAttribute attrib3 = new ChartAttribute(Colors.Yellow, 3, ChartObj.LS_SOLID,
Colors.Yellow);
ChartAttribute attrib4 = new ChartAttribute(Colors.MediumPurple, 2, ChartObj.LS_DOT_1_4,
Colors.MediumPurple);
AntennaLineMarkerPlot thePlot1 = new AntennaLineMarkerPlot(pAntennaTransform, Dataset1,
attrib1);
chartVu.AddChartObject(thePlot1);

AntennaLineMarkerPlot thePlot2 = new AntennaLineMarkerPlot(pAntennaTransform, Dataset2,
attrib2);
chartVu.AddChartObject(thePlot2);

AntennaAnnotation thePlot3 = new AntennaAnnotation(pAntennaTransform,
ChartObj.ANTENNA_ANNOTATION_ANGULAR, 180, attrib3);
chartVu.AddChartObject(thePlot3);

AntennaAnnotation thePlot4 = new AntennaAnnotation(pAntennaTransform,
ChartObj.ANTENNA_ANNOTATION_RADIUS, 12, attrib4);
chartVu.AddChartObject(thePlot4);
```

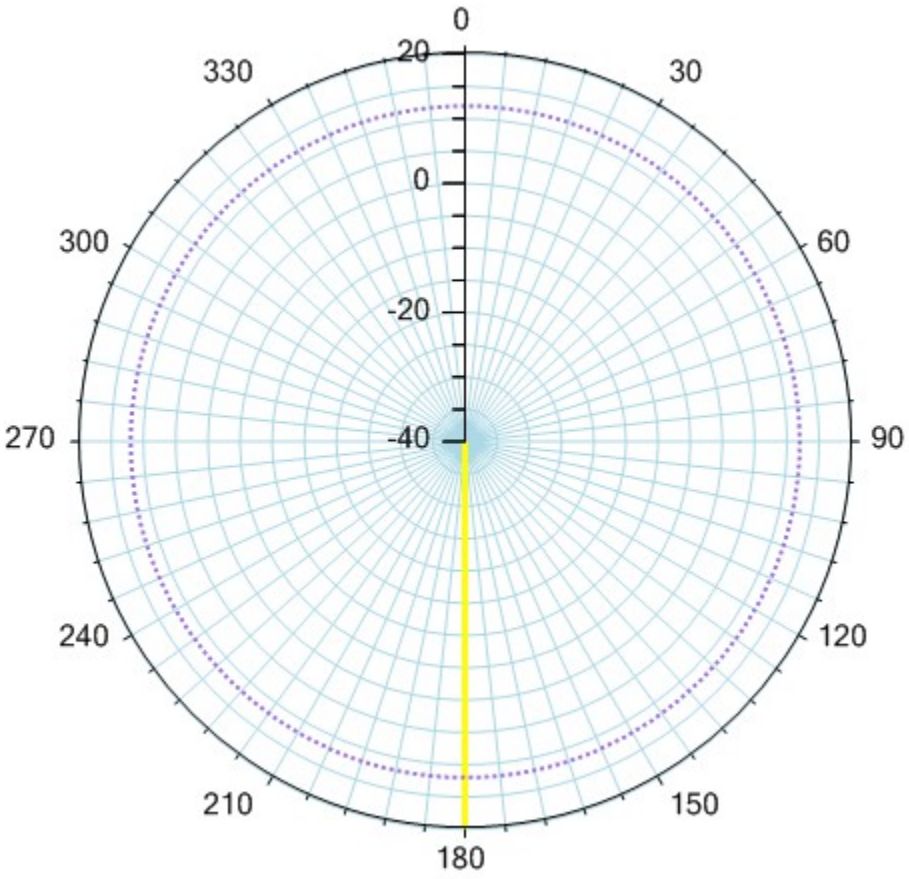
Class AntennaAnnotation

GraphObj

|

+--AntennaAnnotation

The **AntennaAnnotation** class is used to highlight either a specific radius or angular value in an antenna chart. The radius is highlighted using a circle, and the angular value is highlighted using a line draw from the origin to the outer edge of the antenna chart.



Two annotations – a radius annotations (dotted blue line) at the radial value 12, and an angular annotations (solid yellow line) at the angular value 180 degrees.

AntennaAnnotationPlot constructor

```
[C#]
public AntennaAnnotation(
   AntennaCoordinates transform,
   int annotationtype,
   double value,
   ChartAttribute attrib
);
```

transform The coordinate system for the new **AntennaScatterPlot** object.

annotationtype The annotation type. Use one of the annotation type constants: ANTENNA_ANNOTATION_ANGULAR (draws a radial line at the specified angular value, from the orign to the outer edge of the antenna chart, or ANTENNA_ANNOTATION_RADIUS (draws a circle at the specified radius value).

value	The value of the annotation. For an angular annotation, specify the value in degrees. For a radial annotation, specify a value within the range of the antenna minimum and maximum radial values.
attrib	Specifies the attributes (size, line and fill color) for the annotation.

See previous example for a programming example using AntennaAnnotationPlot.

19. Legends

Legend
StandardLegend
BubblePlotLegend

Charts containing multiple chart objects, line plots, bar graphs and scatter plots for example, usually require a legend. The legend provides a key so that the viewer of the chart can figure out what data is associated with what chart object. The bounding box of the legend is rectangular and can reside anywhere in the chart window: inside the plot area, overlapping it or completely outside. The legend rectangle can have a border and can be filled with a solid color or left transparent. The legend object can hold one or more legend items, where each legend item is a symbol-text string combination providing the key for one of the plot objects in the graph. The legend can also have a title and footer.

The **Legend** class is the abstract base class for chart legends. It organizes a collection of legend items as a rectangular object.

The **StandardLegend** is a concrete implementation of the **Legend** class and it is the primary legend class for all plot objects except for bubble plots. The legend items objects display in a row or column format. Each legend item contains a symbol and descriptive string. The symbol normally associates the legend item to a particular plot object, and the descriptive string describes what the plot object represents.

The **BubblePlotLegend** is a concrete implementation of the **Legend** class and it is the legend class for bubble plots. The legend items objects display as offset, concentric circles with descriptive text giving the key for the value associated with a bubble of this size.

Standard Legends

Class StandardLegend

GraphObj
|
+-- StandardLegend

The **StandardLegend** is the primary legend class for all plot objects except for bubble plots. The class manages a list of **LegendItems** that holds the symbols and descriptive text for the symbols.

StandardLegend constructors

```
[C#]
public StandardLegend(
```

```
   double rx,
   double ry,
   ChartAttribute attrib,
   int nlayoutlmode
);

public StandardLegend(
   double rx,
   double ry,
   double rwidth,
   double rheight,
   ChartAttribute attrib,
   int nlayoutlmode
);
```

rx	The x-position, in chart normalized coordinates, of the legend rectangle.
ry	The y-position, in chart normalized coordinates, of the legend rectangle.
rwidth	The width, in chart normalized coordinates, of the legend rectangle.
rheight	The height, in chart normalized coordinates, of the legend rectangle.
attrib	Specifies the outline color, outline line style, and fill color for the legend rectangle.
nlayoutmode	Specifies if the legend has a horizontal, or vertical layout. Use one of the orientation constants: HORIZ_DIR (row major) or VERT_DIR (column major).

Add legend items to a legend using one of the AddLegendItem methods.

AddLegendItem methods

[C#]

```
public int AddLegendItem(
   string stext,
   int nsymbol,
   ChartAttribute attrib,
   ChartFont thefont
);

public int AddLegendItem(
   string stext,
   PathGeometry symbolshape,
   ChartAttribute attrib,
   ChartFont thefont
);

public int AddLegendItem(
   LegendItem legenditem
);
```

```
public int AddLegendItem(
   string stext,
   int nsymbol,
   GraphObj chartobj,
   ChartFont thefont
);

public int AddLegendItem(
   string stext,
   PathGeometry symbolshape,
   GraphObj chartobj,
   ChartFont thefont
);

public int AddLegendItem(
   string stext,
   int nsymbol,
   ChartPlot chartobj,
   int ngroup,
   ChartFont thefont
);
```

stext	Specifies the text string for the legend item.
nsymbol	Specifies the symbol for the legend item. Use one of the chart symbol constants: NOSYMBOL, SQUARE, TRIANGLE, DIAMOND, CROSS, PLUS, STAR, LINE, HBAR, VBAR, or CIRCLE.
chartobj	The color and fill attributes for the legend item are copied from the attributes of this **ChartPlot** object.
symbolshape	Specifies a user defined shape to use as the legend item symbol.
attrib	Specifies the **ChartAttribute** object to get the color and fill attributes of the legend item.
thefont	Specifies the text font for the legend item.

The AddLegendItem returns the current number of legend items.

Simple legend example (extracted from the example program PolarCharts, class PolarLineFillAndScatterChart)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
SimpleLinePlot thePlot1 =
   new SimpleLinePlot(pTransform1, Dataset1, attrib1);
```

```
chartVu.AddChartObject(thePlot1);
.
.
.
SimpleLinePlot thePlot2 =
    new SimpleLinePlot(pTransform1, Dataset2, attrib2);
chartVu.AddChartObject(thePlot2);
.
.
.
SimpleLinePlot thePlot3 =
       new SimpleLinePlot(pTransform1, Dataset3, profitAttrib);
.
.
.
ChartFont legendFont = new ChartFont("Microsoft Sans Serif", 14, FontStyle.Normal,
FontWeights.Bold);
ChartAttribute legendAttributes =
    new ChartAttribute(Colors.Gray, 1,
    ChartObj.LS_SOLID, Color.FromArgb(255,155,155,155));
legendAttributes.SetFillFlag(true);
legendAttributes.SetLineFlag(true);
StandardLegend legend =
       new StandardLegend(0.2, 0.15, 0.3, 0.3,
           legendAttributes, StandardLegend.VERT_DIR);
legend.AddLegendItem("Expenses",ChartObj.LINE, thePlot1, legendFont);
legend.AddLegendItem("Revenue", ChartObj.LINE, thePlot2, legendFont);
legend.AddLegendItem("Profit", ChartObj.HBAR, thePlot3, legendFont);
legend.AddLegendItem("Loss", ChartObj.HBAR, lossAttrib, legendFont);

chartVu.AddChartObject(legend);
```

Bubble Plot Legends

Class BubblePlotLegend

GraphObj

|

+-- BubblePlotLegend

The **BubblePlotLegend** is the primary legend class for bubble plots. The class manages a list of **BubblePlotLegendItem** that holds the symbols and descriptive text for the symbols.

BubblePlotLegend constructors

```
[C#]
public BubblePlotLegend(
   BubblePlot plot,
   double rx,
   double ry,
   double rwidth,
   double rheight,
   ChartAttribute attrib
);


public BubblePlotLegend(
   BubblePlot plot,
   double rx,
   double ry,
```

```
   ChartAttribute attrib
);
```

plot	The bubble plot object the legend is associated with.
rx	The x-position, in chart normalized coordinates, of the legend rectangle.
ry	The y-position, in chart normalized coordinates, of the legend rectangle.
rwidth	The width, in chart normalized coordinates, of the legend rectangle.
rheight	The height, in chart normalized coordinates, of the legend rectangle.
attrib	Specifies the outline color, outline line style, and fill color for the legend rectangle.

Add legend items to a legend using one of the AddLegendItem methods.

AddLegendItem methods

```
[C#]
public int AddLegendItem(
   string stext,
   double rsize,
   ChartAttribute attrib,
   ChartFont thefont
);

public BubblePlotLegend(
   BubblePlot plot,
   double rx,
   double ry,
   double rwidth,
   double rheight,
   ChartAttribute attrib
);
```

stext	Specifies the text string for the legend item.
rsize	Specifies the size of the bubble for this item, in the same units as the coordinate system the bubble plot is placed in.
chartobj	The color and fill attributes for the legend item are copied from the attributes of this chart object.
thefont	Specifies the text font for the legend item.

The method returns the current number of legend items.

Simple legend example (extracted from the example program ScatterPlots, class BubbleChart)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
ChartAttribute attrib1 = new ChartAttribute (Colors.Black, 0,ChartObj.LS_SOLID);
attrib1.SetFillColor (Color.FromArgb(255,177, 33, 33));
attrib1.SetFillFlag (true);
BubblePlot thePlot1 = new BubblePlot(pTransform1, Dataset1,
       ChartObj.SIZE_BUBBLE_RADIUS, attrib1);
chartVu.AddChartObject(thePlot1);

ChartFont theTitleFont = new ChartFont("Microsoft Sans Serif", 14, FontStyle.Normal,
FontWeights.Bold);
ChartTitle mainTitle = new ChartTitle(pTransform1, theTitleFont,
       "DOT COM Bankruptcies and CEO Compensation");
mainTitle.SetTitleType(ChartObj.CHART_HEADER);
mainTitle.SetTitlePosition( ChartObj.CENTER_GRAPH);
chartVu.AddChartObject(mainTitle);

ChartFont theFooterFont = new ChartFont("Microsoft Sans Serif", 10,  FontStyle.Normal,
FontWeights.Bold);
ChartTitle footer = new ChartTitle(pTransform1, theFooterFont,
"The size (radius or area) of the bubble adds an additional dimension to the graph.");

ChartAttribute attrib2 =
       new ChartAttribute (Color.FromArgb(255,177, 33, 33), 0,ChartObj.LS_SOLID);
attrib1.SetFillColor (Color.FromArgb(255,177, 33, 33));
ChartFont legendFont = new ChartFont("Microsoft Sans Serif", 10, FontStyle.Normal);
ChartAttribute legendAttributes =
       new ChartAttribute(Colors.Black, 1,ChartObj.LS_SOLID, Colors.White);
legendAttributes.SetFillFlag(true);
legendAttributes.SetLineFlag(true);

BubblePlotLegend legend =
       new BubblePlotLegend(thePlot1, 0.82, 0.15, 0.14, 0.25, legendAttributes);
legend.AddLegendItem("$10 Million",10, attrib2, legendFont);
legend.AddLegendItem("$25 Million", 25, attrib2, legendFont);
legend.AddLegendItem("$40 Million", 40, attrib2, legendFont);
legend.AddLegendGeneralText(ChartObj.LEGEND_HEADER,
           "Bubble Size", Colors.Black, legendFont);
chartVu.AddChartObject(legend);
```

20. Text Classes

ChartText
- **ChartTitle**
- **AxisTitle**
- **ChartLabel**
  - **StringLabel**
  - **TimeLabel**
  - **NumericLabel**
  - **ElapsedTimeLabel**

The software uses the **ChartText** classes to position and format text in a chart. Examples of classes derived from the **ChartText** include the **ChartLabel**, **AxisLabels**, **ChartTitle**, and **AxisTitle** classes. The **Legend, PieChart** and **ChartPlot** classes, while not derived from the text classes, use them internally.

Simple Text Classes

Class ChartText

GraphObj
|
+--ChartText

The **ChartText** class is the base class for all text output classes. The **ChartText** class formats and places text in a chart. Position the **ChartText** objects using any of the coordinate systems. Rotate and justify the text vertically and horizontally. Insert a CR (carriage return, ASCII 13) character at line breaks for multiline text. The most common constructors are:

ChartText constructors

```
[C#]
public ChartText(
   PhysicalCoordinates transform,
   ChartFont tfont,
   string tstring,
   double x,
   double y,
   int npostype,
   int xjust,
   int yjust,
   int rotation
);

public ChartText(
   PhysicalCoordinates transform,
   ChartFont tfont,
   string tstring,
   double x,
   double y,
```

```
   int npostype
);
```

transform	Places the text in the coordinate system defined by transform.
tfont	A reference to a **ChartFont** object.
Tstring	A reference to a string object.
x	Specifies the x-value of the text position
y	Specifies the y-value of the text position
npostype	Specifies the if the position of the text is specified in physical coordinates, normalized coordinates or UWP device coordinates. Use one of the position constants: DEV_POS, PHYS_POS, NORM_GRAPH_POS, NORM_PLOT_POS.
xjust	Specifies the horizontal justification of the text. Use one of the text justification constants: JUSTIFY_MIN, JUSTIFY_CENTER or JUSTIFY_MAX.
yjust	Specifies the vertical justification of the text. Use one of the text justification constants: JUSTIFY_MIN, JUSTIFY_CENTER or JUSTIFY_MAX.
rotation	The rotation (-360 to 360 degrees) of the text in the normal viewing plane.

Place text in is a time coordinate system (**TimeCoordinates**) by converting the time x-position to milliseconds and using the milliseconds as the x-position value.

ChartText example (extracted from the example program MultiLinePlots, class Multilines)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
ChartFont theLabelFont = new ChartFont("Microsoft Sans Serif", 10, FontStyle.Normal,
FontWeights.Bold);
ChartText currentLabel1 = new ChartText(pTransform1, theLabelFont,
           "I(b) = 50uA", 15.5, y1[0,85]+1 , ChartObj.PHYS_POS);
chartVu.AddChartObject(currentLabel1);

ChartText currentLabel2 = new ChartText(pTransform1, theLabelFont,
               "I(b) = 100uA", 15.5, y1[1,85]+1 , ChartObj.PHYS_POS);
chartVu.AddChartObject(currentLabel2);

ChartText currentLabel3 = new ChartText(pTransform1, theLabelFont,
           "I(b) = 150uA",15.5, y1[2,85]+1 , ChartObj.PHYS_POS);
chartVu.AddChartObject(currentLabel3);
```

```
ChartText currentLabel4 = new ChartText(pTransform1, theLabelFont,
           "I(b) = 200uA", 15.5, y1[3,85]+1 , ChartObj.PHYS_POS);
chartVu.AddChartObject(currentLabel4);

ChartText currentLabel5 = new ChartText(pTransform1, theLabelFont,
           "I(b) = 250uA", 15.5, y1[4,85]+1 , ChartObj.PHYS_POS);
chartVu.AddChartObject(currentLabel5);
```

ChartText time coordinates example (extracted from the example program MiscCharts, class LineGap)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.

ChartFont theLabelFont = new ChartFont("Microsoft Sans Serif", 14, FontStyle.Normal,
FontWeights.Bold);
ChartText chartLabel1 = new ChartText(pTransform1,
           theLabelFont, "Sales", xValues[1].GetCalendarMsecs(),
           groupBarData[1,1], ChartObj.PHYS_POS);
chartLabel1.SetColor(Colors.White);
chartLabel1.SetYJust(ChartObj.AXIS_MIN);
chartVu.AddChartObject(chartLabel1);
```

Chart Title Classes

Class ChartTitle

ChartText

|

+--ChartTitle

The **ChartTitle** class creates a header, subheader or footer for a chart. The most common constructors are:

ChartTitle constructors

```
[C#]
public ChartTitle(
   PhysicalCoordinates transform,
   ChartFont tfont,
   string tstring
);

public ChartTitle(
   PhysicalCoordinates transform,
   ChartFont tfont,
   string tstring,
   int ntitletype,
   int ntitlepos
);
```

transform	Places the text in the coordinate system defined by transform.
tfont	A reference to a **ChartFont** object.
tstring	A reference to a string object.
ntitletype	The title can be a header, subhead or footer. Use one of the title type constants: CHART_HEADER, CHART_SUBHEAD or CHART_FOOTER.
ntitlepos	The title can be centered with respect to the entire graph area, or the plot area. Use one of the title position constants: CENTER_GRAPH or CENTER_PLOT.

ChartTitle example (extracted from the example program SimpleLinePlots, class LineFill)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
ChartFont theTitleFont = new ChartFont("Microsoft Sans Serif", 16, FontStyle.Normal,
FontWeights.Bold);
mainTitle = new ChartTitle(pTransform1, theTitleFont, "Profits are Expected to Rise");
mainTitle.SetTitleType(ChartObj.CHART_HEADER);
mainTitle.SetTitlePosition( ChartObj.CENTER_GRAPH);
mainTitle.SetColor(Colors.White);
chartVu.AddChartObject(mainTitle);
ChartFont theFooterFont = new ChartFont("Microsoft Sans Serif", 9, FontStyle.Normal,
FontWeights.Bold);
footer = new ChartTitle(pTransform1, theFooterFont,
    "Graphs can have background gradients, semi-transparent colors, legends, titles and
data tooltips.");
footer.SetTitleType(ChartObj.CHART_FOOTER);
footer.SetTitlePosition( ChartObj.CENTER_GRAPH);
footer.SetTitleOffset(8);
footer.SetColor(Colors.White);
chartVu.AddChartObject(footer);
```

Class AxisTitle

ChartText

|

+--AxisTitle

The **AxisTitle** class creates a title for an axis. The text is horizontal for x-axis titles and vertical for y-axis titles. The most common constructor is:

AxisTitle Constructor

```
[C#]
public AxisTitle(
   Axis axis,
   ChartFont thefont,
   string s
);
```

axis	The base axis this title is associated with.
thefont	The font object used to display the axis title.
s	Sets the title string.

ChartTitle example (extracted from the example program ScatterPlots, class LabeledDatapoints)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
LinearAxis yAxis = new LinearAxis(pTransform1, ChartObj.Y_AXIS);
chartVu.AddChartObject(yAxis);

NumericAxisLabels xAxisLab = new NumericAxisLabels(xAxis);
xAxisLab.SetTextFont(theFont);
chartVu.AddChartObject(xAxisLab);

NumericAxisLabels yAxisLab = new NumericAxisLabels(yAxis);
yAxisLab.SetTextFont(theFont);
chartVu.AddChartObject(yAxisLab);

ChartFont titleFont = new ChartFont("Microsoft Sans Serif", 12, FontStyle.Normal,
FontWeights.Bold);
AxisTitle yaxistitle = new AxisTitle( yAxis, titleFont, "Test Score");
chartVu.AddChartObject(yaxistitle);

AxisTitle xaxistitle = new AxisTitle( xAxis, titleFont, "Student #");
chartVu.AddChartObject(xaxistitle);
```

Numeric, Time, Elapsed Time and String Label Classes

Class ChartLabel

ChartText
|
+-- ChartLabel
|
+--StringLabel
|
+--TimeLabel

|
+--ElapsedTimeLabel
|
+--NumericLabel

The **ChartLabel** class is the abstract base class for all of the formatted label classes. The axis label classes use formatted labels to label the axis tick marks. They are also useful for chart annotations. Position the objects using any of the coordinate systems. Rotate and justify the text vertically and horizontally.

NumericLabel constructors

```
[C#]
public NumericLabel(
   PhysicalCoordinates transform,
   ChartFont tfont,
   double initialvalue1,
   double x,
   double y,
   int npostype,
   int nnumformat,
   int ndecimal
);


public NumericLabel(
   PhysicalCoordinates transform,
   ChartFont tfont,
   double initialvalue1,
   double x,
   double y,
   int npostype,
   int nnumformat,
   int ndecimal,
   int xjust,
   int yjust,
   double rotation
);
```

transform	Places the text in the coordinate system defined by transform.
tfont	A reference to a ChartFont object.
initialvalue	The initial value of the numeric label.
x	Specifies the x-value of the text position
y	Specifies the y-value of the text position
npostype	Specifies the if the position of the text is specified in physical coordinates, normalized coordinates or UWP device coordinates. Use one of the position constants:

	DEV_POS, PHYS_POS, NORM_GRAPH_POS, NORM_PLOT_POS.
nnumformat	Specifies the numeric format of the label. Use one of the numeric format constants : DECIMALFORMAT, SCIENTIFICFORMAT, BUSINESSFORMAT, ENGINEERINGFORMAT, PERCENTFORMAT, CURRENCEYFORMAT and EXPONENTFORMAT.
ndecimal	The number of digits to display to the right of the decimal point.
xjust	Specifies the horizontal justification of the text. Use one of the text justification constants: JUSTIFY_MIN, JUSTIFY_CENTER or JUSTIFY_MAX.
yjust	Specifies the vertical justification of the text. Use one of the text justification constants: JUSTIFY_MIN, JUSTIFY_CENTER or JUSTIFY_MAX.
rotation	The rotation (-360 to 360 degrees) of the text in the normal viewing plane.

The **TimeLabel, ElapsedTimeLabel** and **StringLabel** classes are similar, unique properties for each are listed below.

TimeLabel constructors

```
[C#]
public TimeLabel(
   PhysicalCoordinates transform,
   ChartCalendar date,
   int timeformat
);

public TimeLabel(
   PhysicalCoordinates transform,
   ChartFont tfont,
   ChartCalendar date,
   double x,
   double y,
   int npostype,
   int timeformat,
   int xjust,
   int yjust,
   double rotation
);
```

date	The calendar value used to initialize the label.

timeformat The format used to convert the calendar value to a text string. Use one of the calendar format constants, TIMEDATEFORMAT_XXX.

ElapsedTimeLabel constructors

```
C#
public ElapsedTimeLabel (
   PhysicalCoordinates transform,
   ChartCalendar date,
   int timeformat
);

public ElapsedTimeLabel(
    PhysicalCoordinates transform,
    ChartFont tfont,
    TimeSpan timespan,
    double x,
    double y,
    int npostype,
    int timeformat,
    int xjust,
    int yjust,
    double rotation
)
```

timespan The time span value used to initialize the label.

timeformat The format used to convert the time span value to a text stringUse one of the TIMEDATAFORMAT_ constants: TIMEDATEFORMAT_NONE, TIMEDATEFORMAT_24HMS, TIMEDATEFORMAT_24HM, TIMEDATEFORMAT_MS.

StringLabel constructors

```
[C#]
public StringLabel(
   PhysicalCoordinates transform,
   ChartFont tfont,
   string tstring,
   double x,
   double y,
   int npostype
);

public StringLabel(
   PhysicalCoordinates transform,
   ChartFont tfont,
   string tstring,
   double x,
   double y,
   int npostype,
   int xjust,
```

```
   int yjust,
   double rotation
);
```

tstring A reference to a string object.

Place text in a time coordinate system (**TimeCoordinates**) by converting the time x-position to milliseconds and using the milliseconds as the x-position value.

NumericLabel and StringLabel example (extracted from the example program MouseListeners, class MoveDatapoints)

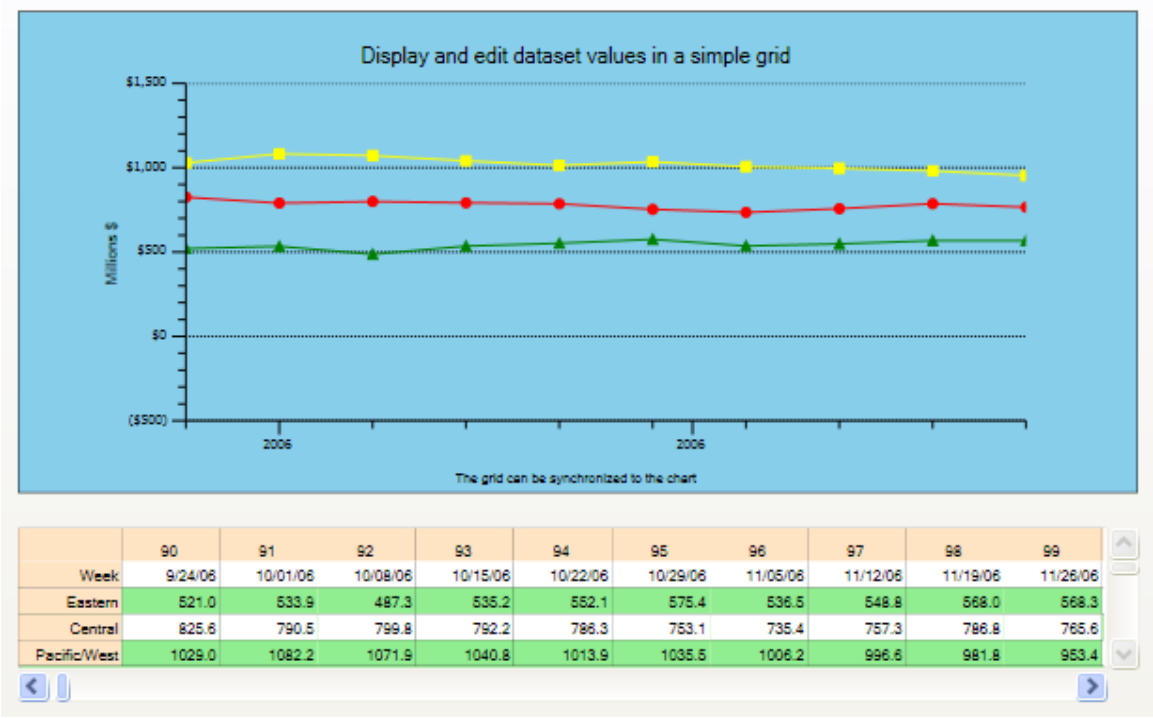
[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
class1Average = Dataset1.GetAverageY();
class2Average = Dataset2.GetAverageY();

StringLabel class1Label =
    new StringLabel(pTransform1, subheadFont, "Class #1" + "\n" +  "Average",
            0.9, 0.3, ChartObj.NORM_GRAPH_POS);
chartVu.AddChartObject(class1Label);

class1AverageLabel =
    new NumericLabel(pTransform1, subheadFont, class1Average,
            0.9, 0.35, ChartObj.NORM_GRAPH_POS, ChartObj.DECIMALFORMAT,1);
chartVu.AddChartObject(class1AverageLabel);

StringLabel class2Label =
    new StringLabel(pTransform1, subheadFont, "Class #2" + "\n" +  "Average",
            0.9, 0.5, ChartObj.NORM_GRAPH_POS);
chartVu.AddChartObject(class2Label);

class2AverageLabel = new NumericLabel(pTransform1, subheadFont, class2Average,
            0.9, 0.55, ChartObj.NORM_GRAPH_POS, ChartObj.DECIMALFORMAT,1);
chartVu.AddChartObject(class2AverageLabel);
```

21. Dataset Viewers

DatasetViewer

Charts and data grids are probably the two most popular ways to display numeric data. At the time of this writing, UWP does not include a data grid control, though you would expect Microsoft to add one at some point. We have created our own grid class that integrates with our chart dataset classes. The **DatasetViewer** can display simple numeric and time-based datasets (**SimpleDataset**, **TimeSimpleDataset, ElapsedTimeSimpleDataset**) and group numeric and time-based datasets (**GroupDataset**, **TimeGroupsDataset, ElapsedTimeGroupDataset**). When a **DatasetViewer** is added to a chart, it can be printed as part of that chart. Background colors, row and column headers, can be customized. The **DatasetViewer** can be scrolled, updated in real-time, and synchronized to the chart, so that scrolling of the DatasetViewer can scroll the chart.



	90	91	92	93	94	95	96	97	98	99
Week	9/24/06	10/01/06	10/08/06	10/15/06	10/22/06	10/29/06	11/05/06	11/12/06	11/19/06	11/26/06
Eastern	521.0	533.9	487.3	535.2	552.1	575.4	536.5	548.8	568.0	568.3
Central	825.6	790.5	799.8	792.2	786.3	753.1	735.4	757.3	786.8	765.6
Pacific/West	1029.0	1082.2	1071.9	1040.8	1013.9	1035.5	1006.2	996.6	981.8	953.4

A DatasetViewer displaying three TimeSimpleDatasets

Class DatasetViewer

ChartView
|
+--DatasetViewer

The **DatasetViewer** is a **ChartView** derived object and as such is an independent **UserControl** object. Use it to view one or more datasets in a chart. Since it is usually not possible or practical to display the entire dataset, the **DatasetViewer** windows a rectangular section of the dataset for display. Scroll bars are used to scroll the rows and columns of the dataset. The **DatasetViewer** constructor defines the size, position, source matrix, the number of rows and columns of the **DatasetViewer** grid, and the starting position of the **DatasetViewer** scrollbar.

In normal use, you add the **DatasetViewer** to the XAML file of a window, just like you have been adding a **ChartView** window. The **DatasetViewer** can be aligned with a chart by placing it in a grid row below or above the chart, or in a grid column to the left or right of the chart. You can allocate a different amount of space for the grid versus the chart using the UWP Grid relative size definition parameters. In the example below, the **ChartView** is place in Grid.Row = 0, where the row has a height of 5*. The **DatasetViewer** is place in GridRow = 1, where the row has a size of 2*. The result is that the chart will occupy 5/7 of the display space, and the dataset viewer 2/7 of the display space. No matter how you resize the window, that relationship will be maintained.

```
<Grid  Background="Gray"  Width="1100">
    <Grid.RowDefinitions>
         <RowDefinition Height="5*"/>
         <RowDefinition Height="2*"/>
    </Grid.RowDefinitions>
    <my:ChartView  Grid.Row="0" Margin="18,11,16,6" Name="simpledatasetviewerchart1"  />
    <my:DatasetViewer  Grid.Row="1"  Margin="18,11,16,6" Name="datasetViewer1"  />
</Grid>
```

DatasetViewer constructor

C#

```
public DatasetViewer(
    ChartView chartvu,
    PhysicalCoordinates transform,
    Rectangle2D posrect,
    ChartDataset dataset,
    int rows,
    int cols,
    int start
)
```

chartvu	The **ChartView** object the **DatasetViewer** is associated with.
transform	The coordinate system the **DatasetViewer** is placed in.
posrect	A positioning rectangle (using normalized chart coordinates) for the dataset viewer, use null if not used.
dataset	A simple, or group, dataset to add to the dataset viewer.
rows	Number of rows to display

cols	Number of columns to display.
start	Starting column of the dataset viewer.

Set unique fonts for the column headers, row headers and grid cells using the ColumnHeaderFont, RowHearderFont and GridCellFont properties.

Turn on the edit feature of the grid cells using the EnableEdit property. Turn on the striped background color of the grid cells using the UseStripedGridBackground property.

Foreground and background attributes of the column headers, row headers and grid cells can be set using the ColumnHeaderAttribute, RowHeaderAttribute, GridAttribute, and AltGridAttribute properties.

You can add multiple datasets to a **DatasetViewer** using the DatasetViewer.AddDataset method. When adding additional datasets, it only adds the y-values of the dataset. It is assumed the x-values of the datasets are the same; otherwise, the columns would lose synchronization.

The row header string for the first grid row, the x-values, is picked up from the first dataset's XString property. If that is null, "X-Values" is displayed for numeric x-values, and "Time" for time-based x-values. Subsequent row header strings, for the y-values, are picked up from the main title string of each associated dataset. In the case of group datasets with multiple y-values for each x-value, row header strings are picked up from the datasets GroupStrings property, which stores one string for each group in the dataset.

You can change the default orientation of the **DatasetViewer** by calling a version of the **DatasetViewer** constructor that has an orientation property as the last parameter. See the NewDemosRev2.VerticalDatasetViewerChart for an example.

Simple DatasetViewer example(extracted from the example program NewDemosRev2 MainPage.xaml, MainPage.xaml.cs and SimpleDatasetViewer)

```
<Grid  Background="Gray" >
     <Grid.RowDefinitions>
           <RowDefinition Height="5*"/>
     <RowDefinition Height="2*"/>
     </Grid.RowDefinitions>
     <my:ChartView  Grid.Row="0" Margin="10,10,10,10" Name="simpledatasetviewerchart1"/>
     <my:DatasetViewer  Grid.Row="1"  Margin="10,10,10,10" Name="datasetViewer1"  />
</Grid>
```

[C#]

```
if (!(double.IsNaN(datasetviewer.Width) && double.IsNaN(datasetviewer.Height)))
    datasetviewer.PreferredSize = new Size(datasetviewer.Width, datasetviewer.Height);
else
    datasetviewer.PreferredSize = new Size(800, 250);
.
.
.
```

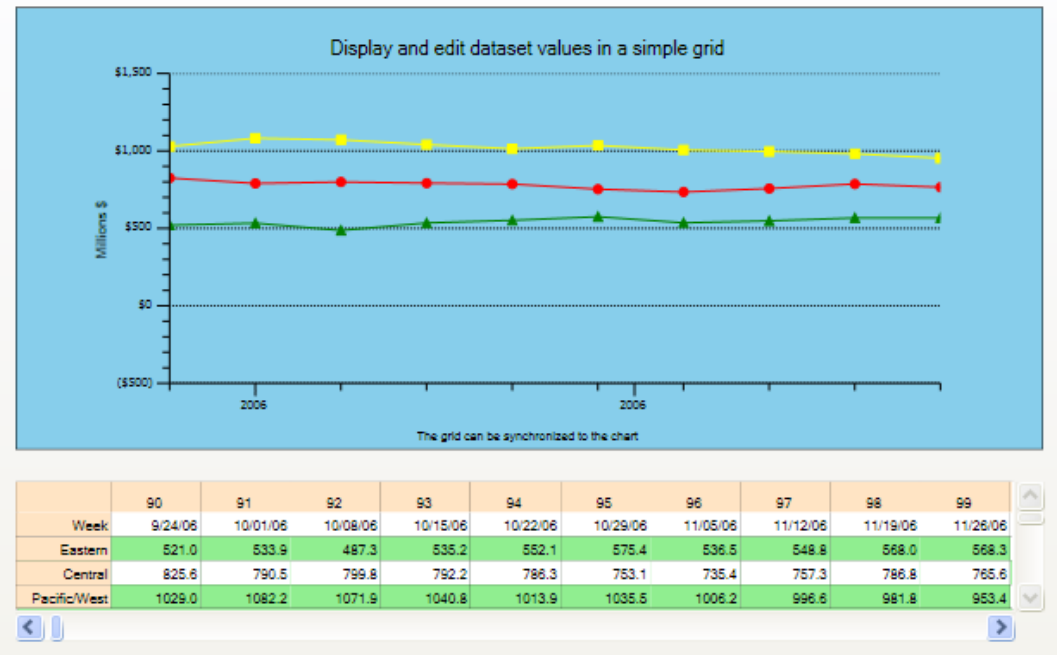
```
Rectangle2D posrect = new Rectangle2D(0.05, 0.5, 0.9, 0.9);
int rows = 4, columns = 10, startindex = initialstartindex;

DatasetViewer.DefaultFont = new ChartFont("Microsoft Sans Serif", 14, FontStyle.Normal);

datasetviewer.InitDatasetViewer(chartVu, pTransform1, null, Dataset1, rows, columns,
startindex);
datasetviewer.AddDataset(Dataset2);
datasetviewer.AddDataset(Dataset3);
datasetviewer.EnableEdit(true);
datasetviewer.DatasetTable.TableBackgroundMode =
ChartGeneralizedTableDisplay.TABLE_SINGLE_COLOR_BACKGROUND_GRIDCELL;
datasetviewer.UseStripedGridBackground = true;
datasetviewer.RowHeaderFont = new ChartFont("Microsoft Sans Serif", 14,FontStyle.Normal);
datasetviewer.ColumnHeaderFont = new ChartFont("Microsoft Sans Serif",
14,FontStyle.Normal);
datasetviewer.GridCellFont = new ChartFont("Microsoft Sans Serif", 14,FontStyle.Normal);
datasetviewer.SyncChart = true;
```

***Note**

Because the example programs use device independent sizing (Stretch) which fills the available space on whatever device is specified, the actual size of the chart windows is usually not known at the time the chart is defined. The result is that that the Width and Height properties of the ChartView, and DatasetViewer are undefined at the time the chart is defined. So all of our examples test for this condition, and set default values for the PreferredSize properties.

```
if (!(double.IsNaN(datasetviewer.Width) && double.IsNaN(datasetviewer.Height)))
    datasetviewer.PreferredSize = new Size(datasetviewer.Width, datasetviewer.Height);
else
    datasetviewer.PreferredSize = new Size(800, 250);
```

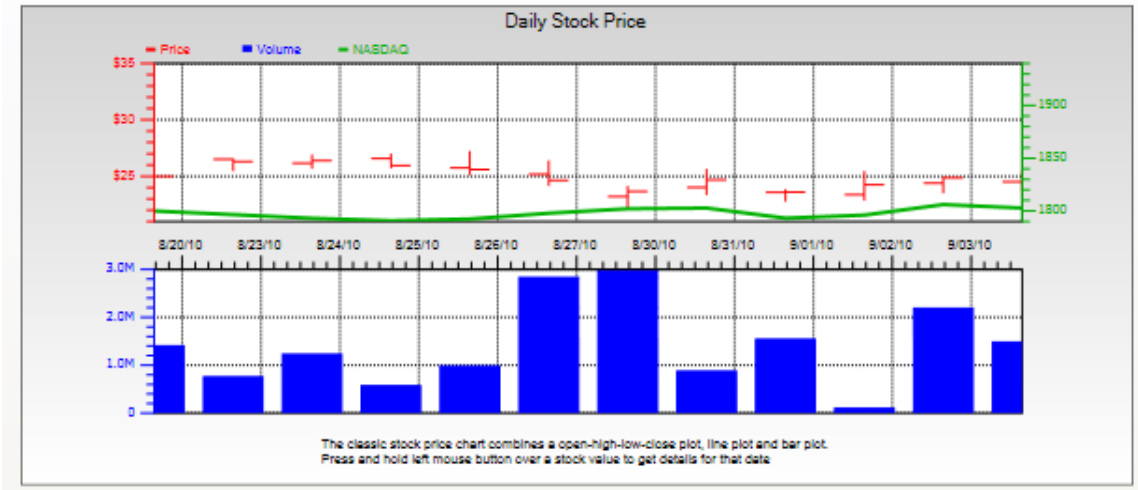
Group DatasetViewer example (extracted from the example program NewDemosRev2 MainPage.xaml, MainPage.xaml.cs and GroupDatasetViewer)



	90	91	92	93	94	95	96	97	98	99
Week	9/24/06	10/01/06	10/08/06	10/15/06	10/22/06	10/29/06	11/05/06	11/12/06	11/19/06	11/26/06
Eastern	521.0	533.9	487.3	535.2	552.1	575.4	536.5	548.8	568.0	568.3
Central	825.6	790.5	799.8	792.2	786.3	753.1	735.4	757.3	786.8	765.6
Pacific/West	1029.0	1082.2	1071.9	1040.8	1013.9	1035.5	1006.2	996.6	981.8	953.4

A **DatasetViewer** displaying a **TimeGroupDataset** display open-high-low-close data.

```
<Grid  Background="Gray" >
     <Grid.RowDefinitions>
        <RowDefinition Height="5*"/>
        <RowDefinition Height="2*"/>
     </Grid.RowDefinitions>
     <my:ChartView  Grid.Row="0" Margin="10,10,10,10" Name="groupdatasetviewer1"  />
     <my:DatasetViewer  Grid.Row="1"  Margin="10,10,10,10" Name="datasetViewer2"  />
</Grid>
```



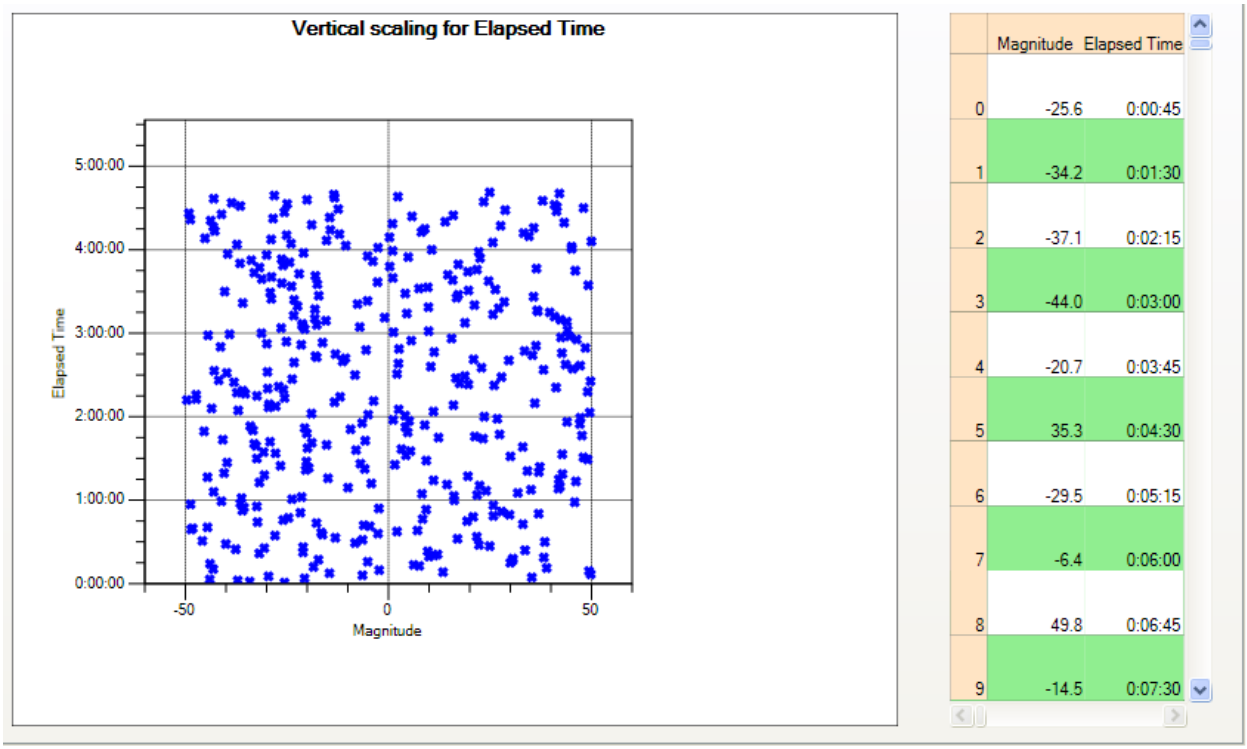
	0	1	2	3	4	5	6	7	8	9	10	11
Day	8/19/10	8/20/10	8/23/10	8/24/10	8/25/10	8/26/10	8/27/10	8/30/10	8/31/10	9/01/10	9/02/10	9/03/10
Open	25.0	26.5	26.2	26.6	25.7	25.2	23.2	24.0	23.6	23.4	24.4	24.5
High	26.0	26.6	26.9	27.0	27.2	26.4	24.1	25.7	23.9	25.5	25.0	25.5
Low	24.0	25.4	25.7	25.7	25.1	24.1	22.0	23.3	22.7	22.8	23.5	23.7
Close	25.0	26.3	26.4	26.0	25.6	24.6	23.7	24.7	23.6	24.2	24.9	23.8

[C#]

```
if (!(double.IsNaN(datasetviewer.Width) && double.IsNaN(datasetviewer.Height)))
    datasetviewer.PreferredSize = new Size(datasetviewer.Width, datasetviewer.Height);
else
    datasetviewer.PreferredSize = new Size(800, 250);
.
.
.

Rectangle2D posrect = new Rectangle2D(0.0, 0.0, 1.0, 1.0);

datasetviewer.InitDatasetViewer(chartVu, pTransform1, posrect, Dataset1, 5, 12, 0);
datasetviewer.EnableEdit(true);
datasetviewer.SyncTableRange = true;
datasetviewer.DatasetTable.TableBackgroundMode =
ChartGeneralizedTableDisplay.TABLE_SINGLE_COLOR_BACKGROUND_GRIDCELL;
datasetviewer.SyncChart = true;
datasetviewer.ResizeMode = ChartObj.AUTO_RESIZE_OBJECTS;
datasetviewer.SetArrayFormat(0, ChartObj.TIMEDATEFORMAT_MDY);
datasetviewer.TransformList.Add(pTransform2);
datasetviewer.TransformList.Add(pTransform3);
```

Vertical Orientation DatasetViewer example (extracted from the example program NewDemosRev2 MainPage.xaml, MainPage.xaml.cs and VerticalDatasetViewerChart)

```
<Grid  Background="Gray" >
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="3*"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <my:ChartView  Grid.Column="0" Margin="10,10,10,10"
          Name="verticaldatasetviewerchart1"/>
    <my:DatasetViewer  Grid.Column="1"  Margin="10,10,10,10" Name="datasetViewer3"  />
</Grid>
```



[C#]

```
if (!(double.IsNaN(datasetviewer.Width) && double.IsNaN(datasetviewer.Height)))
   datasetviewer.PreferredSize = new Size(datasetviewer.Width, datasetviewer.Height);
else
  datasetviewer.PreferredSize = new Size(200, 600);
.
.
.

Rectangle2D posrect = new Rectangle2D(0.0, 0.0, 1.0, 1.0);
int rows = 10, columns = 2, startindex = 0;
datasetviewer.InitDatasetViewer(chartVu, pTransform1, posrect, Dataset1, rows, columns,
startindex, ChartObj.VERT_DIR);
datasetviewer.EnableEdit(true);
datasetviewer.DatasetTable.TableBackgroundMode =
ChartGeneralizedTableDisplay.TABLE_SINGLE_COLOR_BACKGROUND_GRIDCELL;
datasetviewer.UseStripedGridBackground = true;

datasetviewer.PreferredSize = new Size(250, 600);
```

22. Adding Lines, Shapes, Images and Arrows to a Chart

ChartShape
Arrow
ChartImage

It is not possible to take into account every possible graphical object that a programmer wants to add to a graph. Specialized applications require specialized objects. Rather than create a large group of classes that duplicate the functions of the **Arc2D**, **Rectangle2D** and other classes, a generalized class has been created, **ChartShape**, that can place and display in a chart any object that can be expressed as a **Windows.UI.Xaml.Media.PathGeometry**.

The **Arrow** defines an arrow shape useable with the **ChartShape** class. The **Arrow** class creates the arrows in the **ArrowPlot** class, and it can also place individual arrows in a chart. The class creates a base arrow with a custom arrowhead and shaft size. Scale, rotate and position the arrow in a chart.

The **ChartImage** class places a **Windows.UI.Xaml.Media.Imaging.WriteableBitmap** object anywhere in a chart. It can be a small element of the chart, inside or outside of the plot area, or it can be sized to fill the plot area or graph area and used as a background object.

Generic Shape Class

Class ChartShape

GraphObj
|
+--ChartShape

The **ChartShape** class places arbitrary **PathGeometry** objects in a chart. If the shape includes absolute positioning information, use (0,0) as the xy position parameters of the shape. If the shape coordinates are relative coordinates with the object centered on (0,0), place the shape at the position you want using the xy position parameters. The xy position parameters are the rotation origin of shape.

ChartShape constructor

```
[C#]
public ChartShape(
   PhysicalCoordinates transform,
   PathGeometry ashape,
   int shapecoordstype,
   double x,
   double y,
```

```
   int npositiontype,
   int rotation
);
```

transform	The shape object is placed in the coordinate system defined by transform.
ashape	A reference to a **Windows.UI.Xaml.Media.PathGeometry** object.
shapecoordstype	Specifies if the coordinate system defining the shape is specified in physical coordinates, normalized coordinates or UWP device coordinates. Use one of the position constants: DEV_POS, PHYS_POS, NORM_GRAPH_POS, NORM_PLOT_POS.
x	Specifies the x-value of the shape position.
y	Specifies the y-value of the shape position.
npostype	Specifies the if the position of the shape is specified in physical coordinates, normalized coordinates or UWP device coordinates. Use one of the position constants: DEV_POS, PHYS_POS, NORM_GRAPH_POS, NORM_PLOT_POS.
rotation	The rotation, in degrees, of the shape in the normal viewing plane. The rotation will take place about the objects (0.0, 0.0) coordinate. If the object is not defined with a center of (0.0, 0.0) it may be rotated out of the current viewing plane.

ChartShape example (extracted from the example program MultiLinePlots, class MultiLines)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
Color alphaColor =  ChartColor.FromArgb(127,170, 100, 50);
ChartAttribute attrib2 = new ChartAttribute (alphaColor, 1,ChartObj.LS_SOLID, alphaColor);
attrib2.SetFillFlag(true);
Rectangle2D linearRegionRect = new Rectangle2D(0.1,0.1,1.5,50);

PathGeometry rectpath = new PathGeometry();
ChartSupport.AddRectangle(rectpath, linearRegionRect);
ChartShape linearRegionShape = new ChartShape(pTransform1, rectpath,
           ChartObj.PHYS_POS, 0.0, 0.0, ChartObj.PHYS_POS,0);
linearRegionShape.SetChartObjAttributes(attrib2);
chartVu.AddChartObject(linearRegionShape);
```

ChartShape example (extracted from the example program ScatterPlots, class LabeledDatapoints)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
PathGeometry linepath = new PathGeometry();
Point startp = new Point(0.1, 0.1);
Point stopp = new Point(0.9, 0.1);
ChartSupport.AddLineSegment(linepath, startp, stopp);

ChartShape titleLineShape = new ChartShape(pTransform1, linepath,
    ChartObj.NORM_GRAPH_POS, 0.0, 0.0, ChartObj.NORM_GRAPH_POS,0);
titleLineShape.SetLineWidth(3);
chartVu.AddChartObject(titleLineShape);
```

Chart Image Class

Class ChartImage

GraphObj

|

+-- ChartImage

The **ChartImage** class will place a **Windows.UI.Xaml.Media.Imaging.WriteableBitmap** object anywhere in a chart. It can be a small element of the chart, inside or outside of the plot area or it can be sized to fill the plot area or graph area and used as a background object.

ChartImage constructor

```
[C#]
public ChartImage(
   PhysicalCoordinates transform,
   Image aimage,
   double x,
   double y,
   int npostype,
   int rotation
);
```

transform	The coordinate system for the new **ChartImage** object.
aimage	A reference to the **Image** object that is to be placed in the chart.
x	The x-value for the position of the image in the chart.
y	The y-value for the position of the image in the chart.

npostype	Specifies whether the x- and y-position values are specified in normalized coordinates, or physical coordinates. Use one of the position constants: NORM_POS, PHYS_POS.
rotation	The rotation of the image specified in degrees.

Any method (InitializeChart in the code below) which uses the ChartSupport.GetImage method to grab an image bitmap, must be marked async, and the ChartSupport.GetImage call must have the await operator in front of it.

ChartImage example (extracted from the example program ImageCharts, class ImageBackground)

[C#]

```
public async Task InitializeChart()
{
    ChartFont theFont;
    int nnumpnts = 8;
    ChartCalendar []x1= new ChartCalendar[nnumpnts];
    double []y1 = new double[nnumpnts];
    ChartCalendar currentdate = new ChartCalendar(1998, ChartObj.JANUARY, 1);
    int i;

   StorageFolder dataFolder = Windows.Storage.ApplicationData.Current.LocalFolder;
   StorageFolder installationFolder =
Windows.ApplicationModel.Package.Current.InstalledLocation;

.
.
.

   String imagefilename = "ChartClouds.jpg";
   WriteableBitmap aImage =  await ChartSupport.GetImage(installationFolder,
imagefilename);

   if (aImage != null)
   {
       ChartImage chartImage = new ChartImage(pTransform1, aImage,
               0, 0, ChartObj.NORM_GRAPH_POS, 0);

       chartImage.SetSizeMode(ChartObj.COORD_SIZE);
       chartImage.SetImageSize(new Dimension(1, 1));
       chartImage.ZOrder = 20;
       chartVu.AddChartObject(chartImage);
    }
```

Generic Arrow Class

Class Arrow

ChartObj

|
+-- **Arrow**

The **Arrow** defines an arrow shape useable with the **ChartShape** class. The **Arrow** class creates the arrows in the **ArrowPlot** class, and it can also place individual arrows in a chart. The class creates a base arrow with a custom arrowhead and shaft size. Scale, rotate and position the arrow in a chart. The arrow is defined using device coordinates.

Arrow constructor

```
[C#]
public Arrow(
   double arrowshafthalfwidth,
   double arrayshaftlength,
   double arrowheadhalfwidth,
   double arrowheadlength
);
```

arrowshafthalfwidth	Sets the half-width of the arrow shaft. (default 1
arrayshaftlength	Sets the length of the arrow shaft. (default 7)
arrowheadhalfwidth	Sets the half-width of the arrow head. (default 2)
arrowheadlength	Sets the length of the arrow head. (default 3)

The default arrow has a length of about 10 pixels and a width of 4 pixels at the head. The size of the various parts can be set to whatever values you want to create an arrow of with an aspect ratio appropriate to your application. You can scale the arrow by setting the **ArrowScaleFactor** property. Get a **PathGeometry** object defining the arrow shape by calling the **GetArrowShape** method.

Arrow example (extracted from the example program MultiLinePlots, class MultiLines)

[C#]

```
public void InitializeChart(ChartView chartVu)
{
.
.
.
Arrow regionArrow = new Arrow(1,40,6,15);
ChartAttribute arrowAttrib =
       new ChartAttribute (Colors.Black, 1,ChartObj.LS_SOLID, Colors.Black);
arrowAttrib.SetFillFlag(true);
ChartShape arrowShape = new ChartShape(pTransform1, regionArrow.GetArrowShape(),
```

```
        ChartObj.DEV_POS, 1.5, 40.0, ChartObj.PHYS_POS,195);
arrowShape.SetChartObjAttributes(arrowAttrib);
chartVu.AddChartObject(arrowShape);


chartVu.AddChartObject(arrowShape)
```

23. File and Printer Rendering Classes

ChartPrint
BufferedImage

High quality B&W and color printing is an important feature of the charting library. The resulting graph renders on the printer using the resolution of the output device, for both text and graphical elements of the chart, and does not transfer a grainy image from the computer to the printer. The QCChart2D for Windows Apps software uses the **Windows.UI.Xaml.Printing and Windows.Graphics.Printing** services to implement printing. Since the aspect ratio of the printed page is different from the aspect ratio of common displays, options are included that allow different modes for positioning and sizing the chart on the printed page. You can print a couple of different ways. First, you can enable, and select a printer icon, located in the upper left corner of a ChartView window. Second, you can create ChartPrint class in your own code, passing in your own FrameworkElement object, and print from there.

The **BufferedImage** class converts a chart into a **Windows.UI.Xaml.Media.Imaging.RenderedTargetBitmap** object, or saves the chart to a file in a bmp, jpg, gif, tif, png or wmp format. The image file is placeable in a web page or an application program. You can create a "headless" .Net application and render charts without displaying a Windows form, saving the charts as image files.

Print using the ChartView printer icon

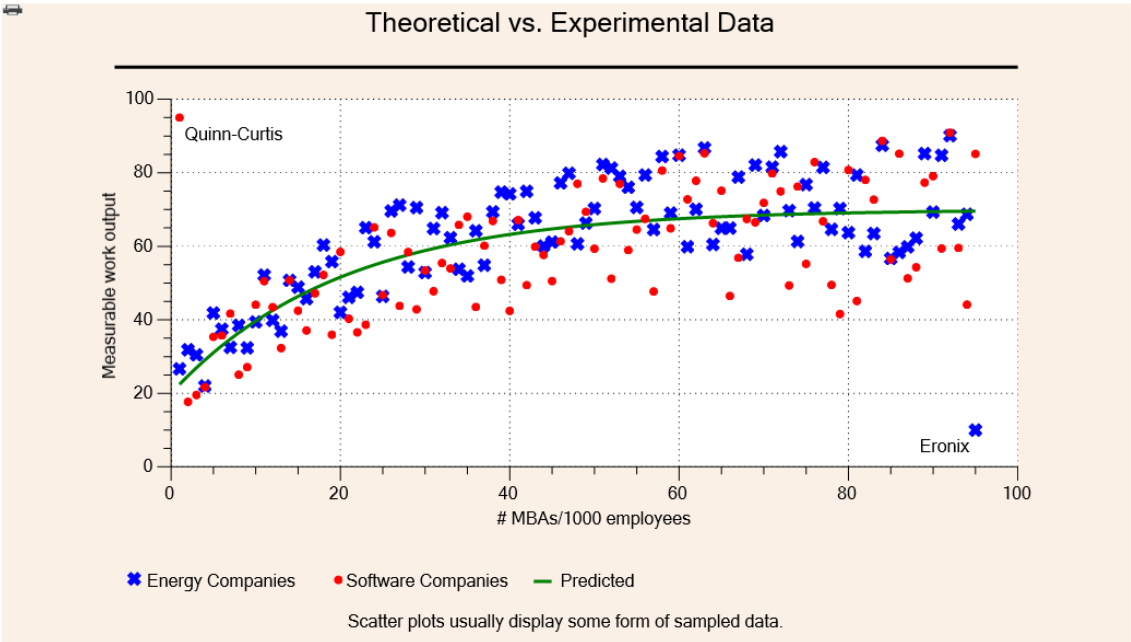
The printer icon is **on** by default. You can turn it off for a given ChartView window by setting the ChartView PrinterButtonVisible property false.

```
chartVu.PrinterButtonVisible = false;
```

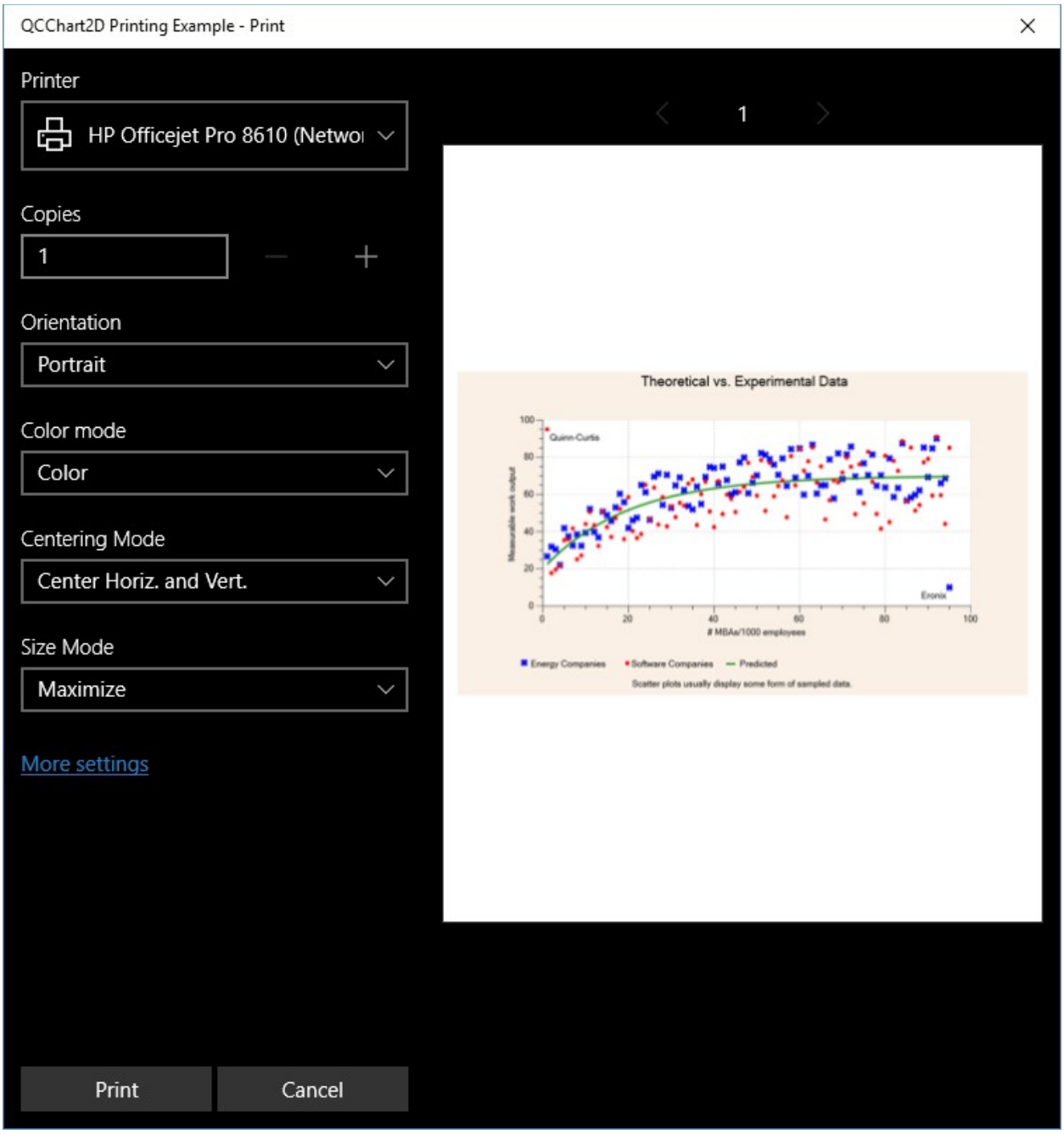
Important Note*
The printer icon image file needs to be included in your projects Assets folder, and in Properties for that file, the **Build Action** should be set to **Content**, and the **Copy to Output Directory** property marked **Copy Always**. You can add the image icon to your project by right clicking on the projects **Assets** folder (from the Solution Explorer) and selecting Add | Existing Item and browse to the Quinn-Curtis\DotNet\lib\Images folder and selecting the **Print_11009.png** file. If the printer icon file is not included in the project, then the printer button will be identified using simple button text which says **Print**.

The printer icon can be placed in one of the four corners of the ChartView. The default is upper left (ChartObj.UPPER_LEFT), but you can also specify ChartObj.UPPER_RIGHT, ChartObj.LOWER_LEFT or ChartObj.LOWER_RIGHT.

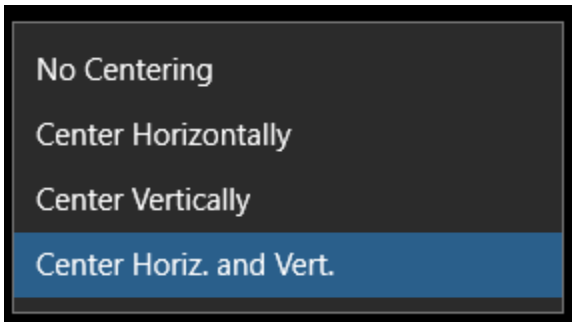
```
spcChart.PrinterButtonVisible = true;
spcChart.PrinterButtonPosition = ChartObj.UPPER_RIGHT;
```



The small icon in the upper left corner of the chart is the print button. Select it and you will invoke the standard Windows App print preview and print dialog.

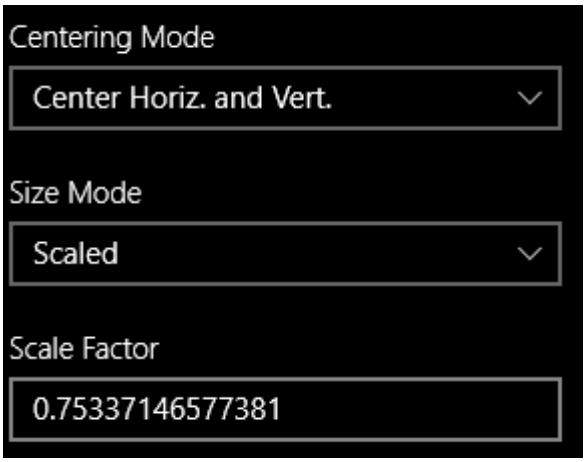


We added Centering Mode and Size Mode options to the standard printer dialog. That way you center the chart horizontally, vertically, both, or neither.



The Size Mode option supports an automatic maximize scaling, explicit scaling, and unscaled (actual size).

If the Scaled option is selected, a scaling factor can be entered.



The charts aspect ratio (W/H) cannot be changed, meaning that the scale factor for the x-dimension and the y-dimension will always be the same.

Regardless of how printing is invoked, either by using the ChartView printer button, or with an explicit call to the ChartPrint class, the Print dialog described above is always displayed. This is a Windows App restriction and we have no way around it.

Printing a Chart

Class ChartPrint

ChartObj

|

+--ChartPrint

The **ChartPrint** class uses uses the **Windows.UI.Xaml.Printing and Windows.Graphics.Printing** services to implement printing. The class selects, setups, and outputs a Xaml FrameworkElement object to a printer. Since our ChartView class is a UserControl, and UserControl is a FrameworkElement, you can directly print a ChartView object. You can also print a FrameworkElement object which contains one or more ChartView objects. We do not support multi-page print preview and printing though. To do that you would have to write your own Printer class, and output multiple pages to it.

ChartPrint constructor

```
public ChartPrint(
        FrameworkElement fe,
        int sizemode
)
```

fe Specifies the **ChartView**, or UWP **Panel** object to be printed.

nsizemode The size mode for the print job. Valid size modes are : ChartObj.PRT_EXACT, ChartObj.PRT_MAX, ChartObj.PRT_SF and ChartObj.PRT_RECT.

Use one of the mapping mode constants:

PRT_MAX Print the view so that paper is used maximally. In portrait mode, the charts width on the printed page will match the short edge of the page, minus a small non-printable margin. In landscape mode the charts width on the printed page will match the long edge of the page, minus a small non-printable margin. The height of the printed chart will be adjusted to keep the proportions of the chart the same. Text prints proportional to other objects, aspect ratio is maintained.

PRT_EXACT	Print the view at the same size as the screen, at least as far as .Net maintains a one to one correspondence in the printing engine. The aspect ratio of the view is maintained.
PRT_RECT	Print the view to the specified rectangle, specified using the SetPrintRect method and normalized coordinates. Regardless of the print rectangle, the aspect ratio of the chart is maintained, so the chart may not fill the entire print rectangle.
PRT_SF	Print the view using the specified scaling factor. Set the scaling factor (> 0) using the ChartPrint TransformScaleFactor property. A scaling factor of 1.0 will produce the same output as the PRT_EXACT mode described above.

When you create the ChartPrint object, it gets a print contract from Windows. From that point on, you just call

```
await Windows.Graphics.Printing.PrintManager.ShowPrintUIAsync();
```

to start the printing process. It must be called using the await operator, because it is asynchronous. If you change the current screen, you need to update the ChartPrint PrintSource object with the current view object on the screen.

printHelper.PrintSource = finPlot1;

See the example PrintChartExample for and example.

Using All of the Paper When Printing

The PRT_MAX mode prints the chart as large as possible, while maintaining the same aspect ratio as the original ChartView. If the width is the limiting factor, the bottom of the printed page will always be blank. The same is true of the PRT_RECT mode. While the PRT_RECT mode can control the size and position of the chart on the printed page, it cannot change the aspect ratio of the chart.

The only way to fill the printed page in portrait or landscape mode is establish the screen ChartView size with the same aspect ratio as the 8 1/2 x 11 printed page printable area (about 6.5 x 9 assuming the 1 inch default margins). Assuming portrait mode, a ChartView sized to 650W x 900H will fill the page, as will other ChartView sizes with the same proportions (500W x 692H, 400W x 554H, 300W x 415 etc.). If you are printing in landscape mode then the chart width and height values would be swapped.

ChartPrint example printing a ChartView, extracted from the example program PrintChartExample, class MainPage.xaml.cs.

[C#]

```
private void PrintMenuItemClick(object sender, RoutedEventArgs e)
{
    int currentChart = flipView1.SelectedIndex;
    if (printHelper == null)
        printHelper = new ChartPrint();

    switch (currentChart)
    {
        case 0:
            printHelper.PrintSource = scatterPlot1;
            PrintChart();
            break;
        case 1:
            printHelper.PrintSource = finPlot1;
            PrintChart();
            break;
        case 2:
            printHelper.PrintSource = barPlot1;
            PrintChart();
            break;
    }
}


private async void PrintChart()
{
    await Windows.Graphics.Printing.PrintManager.ShowPrintUIAsync();

}
```

Capturing the Chart as a Buffered Image

Class BufferedImage

ChartObj

|

+-- BufferedImage

The **BufferedImage** class creates a **RenderTargetBitmap** object that is used to render a **ChartView** object into an image buffer. The rendering takes place when the BufferedImage.Render method or BufferedImage.SaveImage method is called. Inernally, a **BitmapEncoder** object is used to convert the bitmap to a variety of image file formats.

If you want to save a single **ChartView** to a bitmap image, use the constructor which passes in a **ChartView**. If you want to save multiple **ChartView** objects to a single image, use the constructor which passes in a **Panel** object.

BufferedImage constructor

```
public BufferedImage(
        Panel component,
        BufferedImage..::..ImageFormats format
)
```

component	The **ChartView**, or UWP **Panel** object that is the source for the chart image.
imageformat	Specify the image format using one of the BufferedImage.ImageFormats constants: ImageFormats.Jpg, .Jpgx, .Gif, .Png, .Tif, .Bmp .

The **BufferedImage.GetRenderedBitmap** method converts the chart to the **RenderTargetBitmap** object specified by the imageformat object and returns a reference to the resulting bitmap.

BufferedImage example saving a ChartView object (extracted from the example program PrintChartExample, class LineScatterPlotPage1.xaml.cs)

[C#]

```
public async void SaveChartImage()
{
  scatterPlot1.UpdateDraw();
  BufferedImage bimage = new BufferedImage(scatterPlot1);
  String filename = imagefilename + ".jpg";
  StorageFolder localFolder = Windows.Storage.ApplicationData.Current.LocalFolder;

  await bimage.SaveImage(localFolder, filename);
}
```

24. Running the Examples and Using QCChart2D for Windows Apps to Create Universal Windows Apps

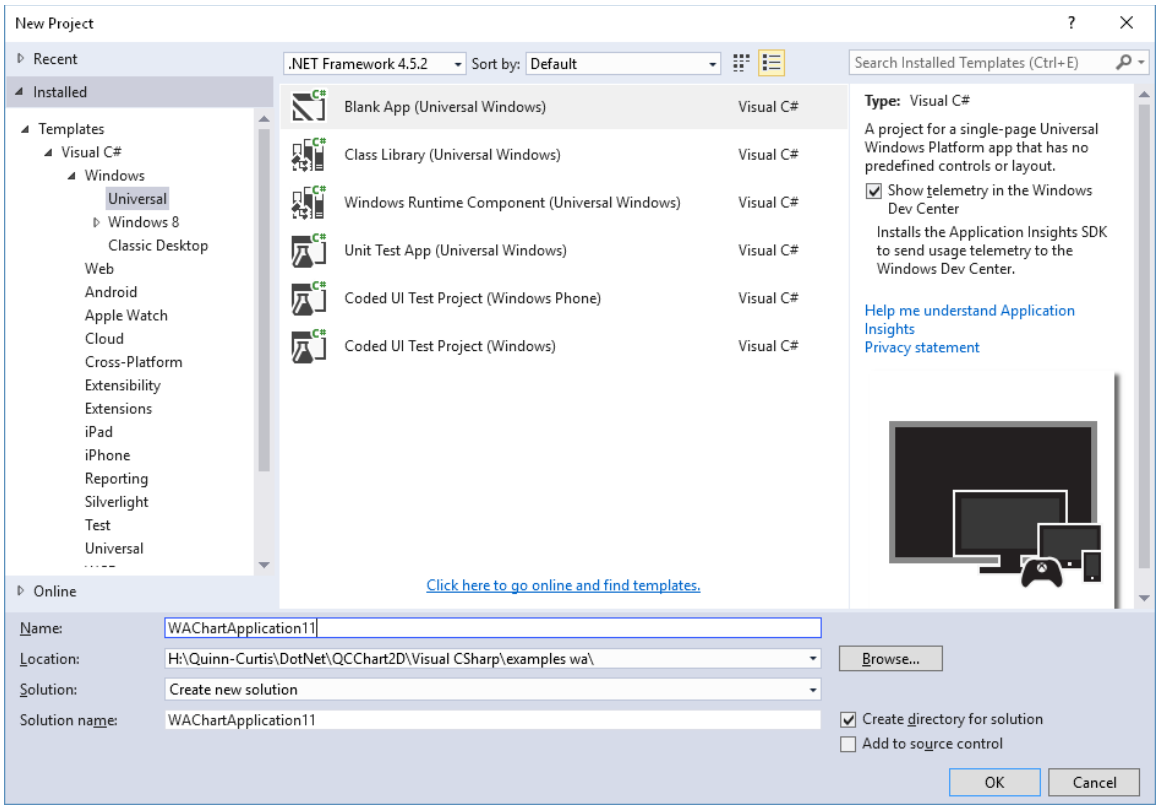
Running the Example Programs

The example programs for **SPC Control Chart Tools for Windows Apps** software are supplied in complete source. In order to save space, they have not been pre-compiled which means that many of the intermediate object files are not present. When initially loaded, at least for the first project you try and load, you may see many errors. But if you Clean and Build the project, the errors will go away.

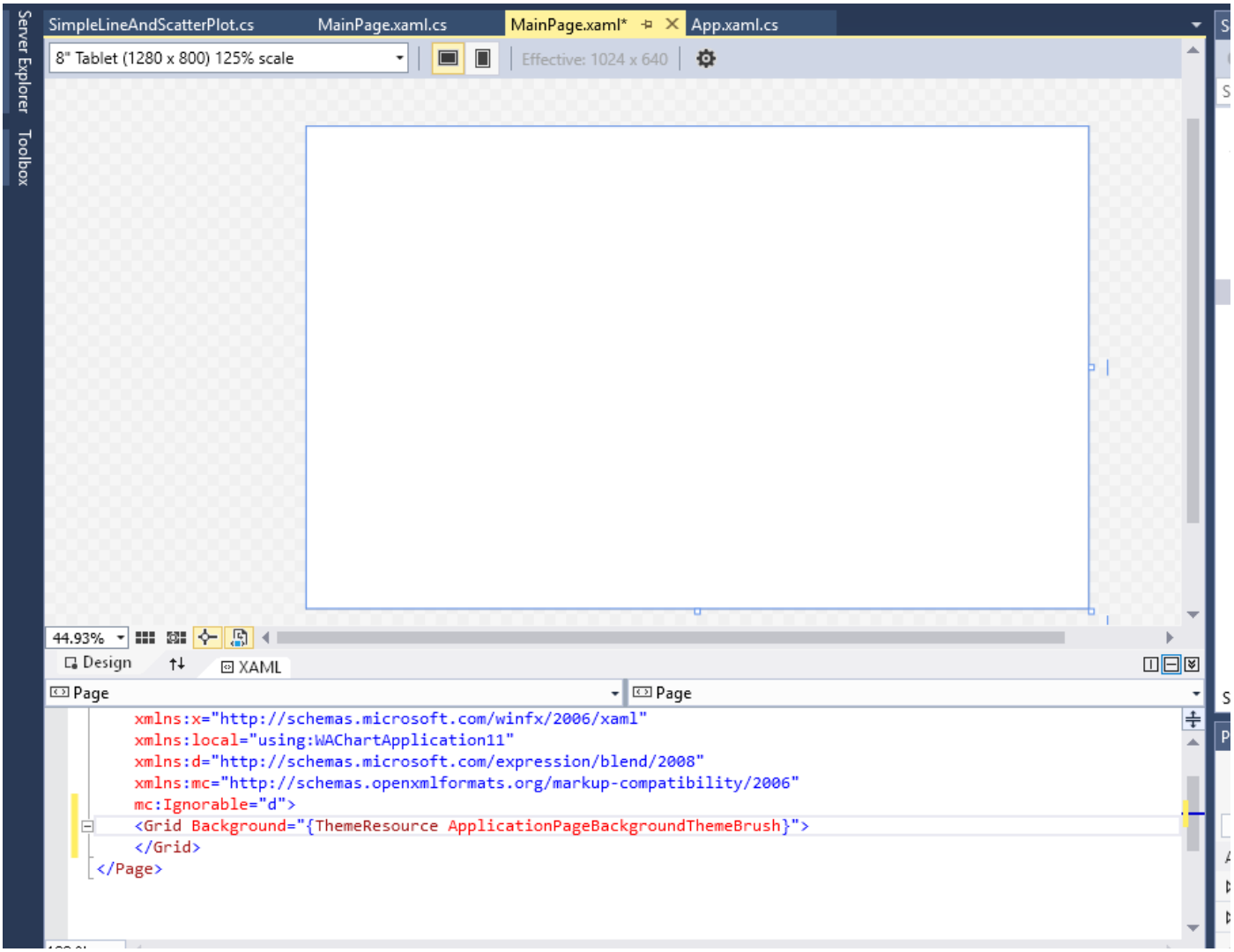
Once you can compile the examples, you should be able to target any of the Universal Windows App platforms you have Visual Studio setup for.

Visual C# for .Net

- If you do not already have an application program project, create one using the Visual Studio project wizard (**File | New | Project .** On the left select a project type of **Templates | Visual C# | Windows | Universal)**. Give the project a unique name, and Browse to specify our Quinn-Curtis\DotNet\QCChart2D\Visual CSharp\examples wa\ folder. In our *examples wa* folder this example is the **WAChartApplication1,** so give your example a different name, such as WAChartApplication11. You will end with a basic UWP based application. For purposes of this example, the chart will placed in the initial, default window.



- The resulting MainPage.xaml page in design mode will something like:



In the upper left hand corner you can change the default target device. We changed it from a 5 inch phone to an 8 inch tablet for this example.

- The XAML portion of the MainPage of the project looks like:

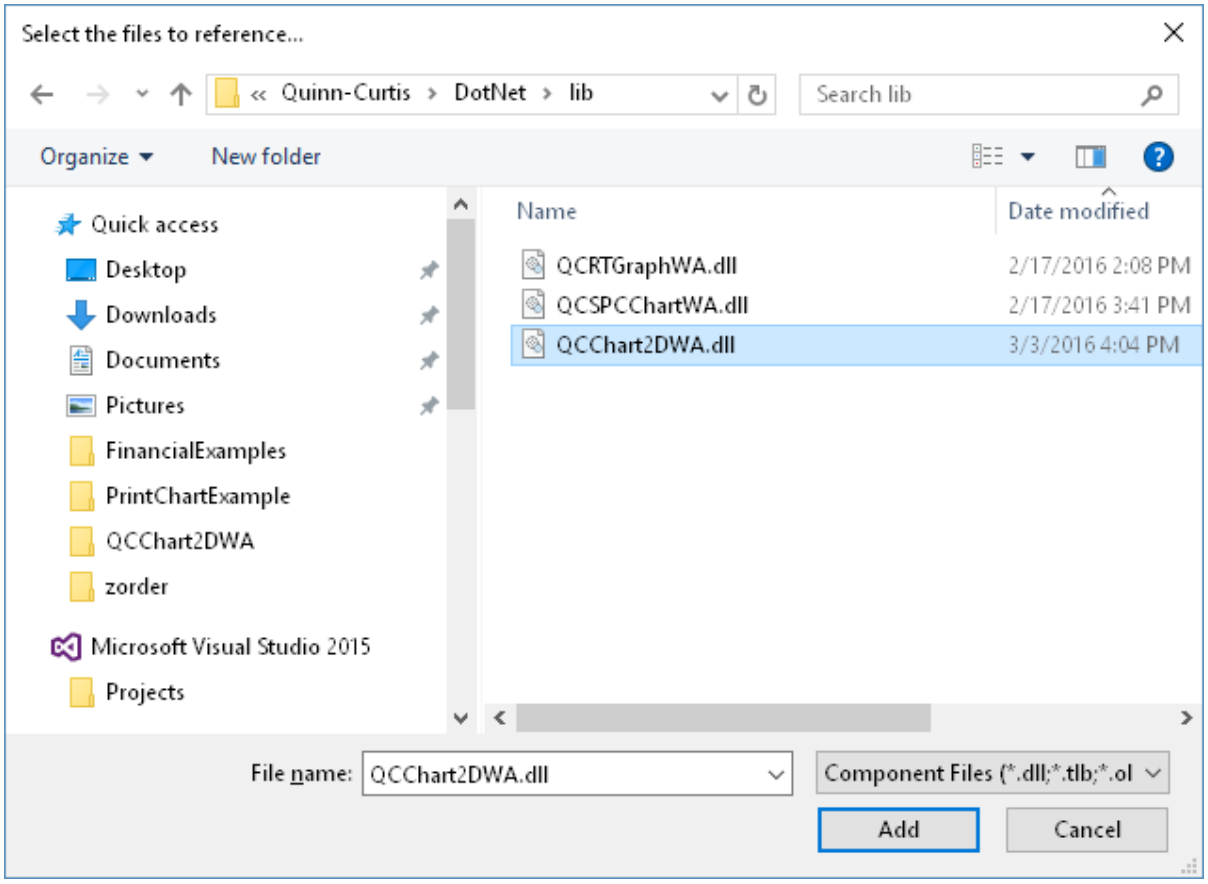
```
<Page
    x:Class="WAChartApplication11.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:WAChartApplication11"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d">

    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">

    </Grid>
</Page>>
```

The window does not yet have any content. First, add a reference to the QCChart2D namespace. In this case the namespace is com.quinncurtis.chart2dwa.

```
xmlns:my="using:com.quinncurtis.chart2dwa"
```

This line needs to be resolved by adding a reference to the QCChart2DWA library to the project.

- Right click on References in the Solution Explorer window and select **Add Reference**. Browse to the Quinn-Curtis/DotNet/lib subdirectory and select the QCChart2DWA.DLL.



- View the **MainPage.xaml** code and add the reference to **ChartView** in the Grid layout panel. The MainPage.xaml file now looks like:

```
<Page
    x:Class="WAChartApplication11.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:WAChartApplication11"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:my="using:com.quinncurtis.chart2dwa"
    mc:Ignorable="d">
    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
```

```
        <my:ChartView x:Name="simplelineandscatterplot1" HorizontalAlignment="Stretch"
Margin="10,10,10,10" VerticalAlignment="Stretch"/>
    </Grid>
</Page>
```

Note that the ChartView object is given the Name *simplelineandscatterplot1*. This name is referenced in the behind code file, MainPage.xaml.cs file. The positioning and size are controlled by the HorizontalAlignment and VerticalAlignment properties, which are set to Stretch, which means that the chart will fill the area of the parent. This allows a large degree of device independence, and works well with grid cells, which can be assigned a relative size to the parent grid. In the case of other containers you may have to assign a specific size.

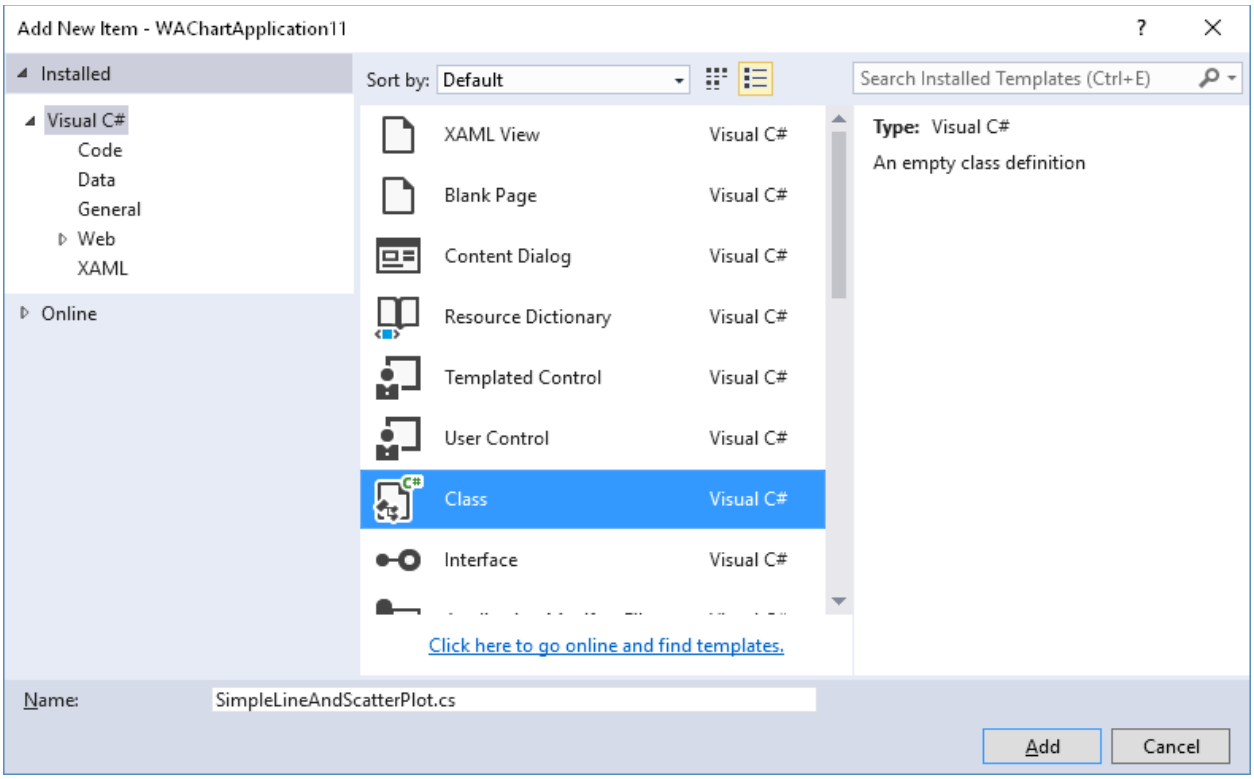
- Display the MainPage.asml.cs behind code file. It will look something like this:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Runtime.InteropServices.WindowsRuntime;
using Windows.Foundation;
using Windows.Foundation.Collections;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Data;
using Windows.UI.Xaml.Input;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Navigation;

// The Blank Page item template is documented at http://go.microsoft.com/fwlink/?
LinkId=402352&clcid=0x409

namespace WAChartApplication11
{
    /// <summary>
    /// An empty page that can be used on its own or navigated to within a Frame.
    /// </summary>
    public sealed partial class MainPage : Page
    {
        public MainPage()
        {
            this.InitializeComponent();
        }
    }
}
```

- Add a reference to the QCChart2DWA namespace, com.quinncurtis.chart2dwa, in the using section of the program.

```
using com.quinncurtis.chart2dwa;
```

- Add a, new, simple class file, named **SimpleLineAndScatterPlot**, to the project. Alternatively, select Add | Existing Item and select the file SimpleLineAndScatterPlot.cs from our WAChartApplication1 example folder. If you do that, make sure you change the declared namespace at the top of the file, namespace WAChartApplication1, to the one your project uses, probably namespace WAChartApplication11.



The resulting SimpleLineAndScatterPlot.cs file will contain:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace WAChartApplication11
{
    class SimpleLineAndScatterPlot
    {
    }
}
```

- Modify the SimpleLineAndScatterPlot file to create the desired chart. Most all of our examples are structured the same way. The constructor is changed to pass in a **ChartView** object. Then a chart initialization routine is called, which adds chart objects to the **ChartView**. This defines the chart. See the SimpleLineAndScatterPlot.cs file of the WAChartApplication1example.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Windows.Foundation;
using Windows.UI;
using Windows.UI.Text;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Input;
using Windows.UI.Xaml.Media;
```

```
using com.quinncurtis.chart2dwa;

namespace WAChartApplication11
{
    public class SimpleLineAndScatterPlot
    {
        public SimpleLineAndScatterPlot(ChartView chartVu)
        {

            InitializeChart(chartVu);
        }

        private void InitializeChart(ChartView chartVu)
        {

            ChartFont theFont;
            int numPoints = 95;
            double[] x1 = new double[numPoints];
            double[] y1 = new double[numPoints];
            double[] y2 = new double[numPoints];
            double[] y3 = new double[numPoints];

            int i;


            // The Width and Height are set in the MainPage.xaml file,
            // so use those values to set the PreferredSize property of the control
            // Make sure they are NOT the NaN value just in case they aren't set when using other code.
            if (!(double.IsNaN(chartVu.Width) && double.IsNaN(chartVu.Height)))
                chartVu.PreferredSize = new Size(chartVu.Width, chartVu.Height);
            else
                chartVu.PreferredSize = new Size(1000, 800);
            for (i = 0; i < numPoints; i++)
            {
                x1[i] = (double)(i + 1);
                y1[i] = 20.0 + 50.0 * (1 - Math.Exp(-x1[i] / 20.0));
                y2[i] = y1[i] + ((20 + 0.2 * x1[i]) * (0.6 - ChartSupport.GetRandomDouble()));
                y3[i] = y1[i] + ((20 + 0.4 * x1[i]) * (0.4 - ChartSupport.GetRandomDouble()));
            }

            y2[94] = 10;
            y3[0] = 95;

            theFont = new ChartFont("Microsoft Sans Serif", 14, FontStyle.Normal);
            SimpleDataset Dataset1 = new SimpleDataset("First", x1, y1);
            SimpleDataset Dataset2 = new SimpleDataset("Second", x1, y2);
            SimpleDataset Dataset3 = new SimpleDataset("Third", x1, y3);

            CartesianCoordinates pTransform1 = new CartesianCoordinates(ChartObj.LINEAR_SCALE,
ChartObj.LINEAR_SCALE);
            pTransform1.AutoScale(Dataset3, ChartObj.AUTOAXES_FAR, ChartObj.AUTOAXES_FAR);

            pTransform1.SetGraphBorderDiagonal(0.15, .15, .90, 0.725);

            Background background = new Background(pTransform1, ChartObj.PLOT_BACKGROUND,
ChartColor.White);

            chartVu.AddChartObject(background);

            LinearAxis xAxis = new LinearAxis(pTransform1, ChartObj.X_AXIS);
            chartVu.AddChartObject(xAxis);

            LinearAxis yAxis = new LinearAxis(pTransform1, ChartObj.Y_AXIS);
            chartVu.AddChartObject(yAxis);

            NumericAxisLabels xAxisLab = new NumericAxisLabels(xAxis);
            xAxisLab.SetTextFont(theFont);
            chartVu.AddChartObject(xAxisLab);

            NumericAxisLabels yAxisLab = new NumericAxisLabels(yAxis);
            yAxisLab.SetTextFont(theFont);
```

```
            chartVu.AddChartObject(yAxisLab);

            ChartFont titleFont = new ChartFont("Microsoft Sans Serif", 14, FontStyle.Normal);
            AxisTitle yaxistitle = new AxisTitle(yAxis, titleFont, "Measurable work output");
            chartVu.AddChartObject(yaxistitle);

            AxisTitle xaxistitle = new AxisTitle(xAxis, titleFont, "# MBAs/1000 employees");
            chartVu.AddChartObject(xaxistitle);

            ChartGrid xgrid = new ChartGrid(xAxis, yAxis, ChartObj.X_AXIS, ChartObj.GRID_MAJOR);
            chartVu.AddChartObject(xgrid);

            ChartGrid ygrid = new ChartGrid(xAxis, yAxis, ChartObj.Y_AXIS, ChartObj.GRID_MAJOR);
            chartVu.AddChartObject(ygrid);

            ChartAttribute attrib1 = new ChartAttribute(ChartColor.Blue, 1, ChartObj.LS_SOLID);
            attrib1.SetFillColor(ChartColor.Blue);
            attrib1.SetFillFlag(true);
            attrib1.SetSymbolSize(15);
            SimpleScatterPlot thePlot1 = new SimpleScatterPlot(pTransform1, Dataset2, ChartObj.STAR,
attrib1);
            chartVu.AddChartObject(thePlot1);

            ChartAttribute attrib2 = new ChartAttribute(ChartColor.Green, 3, ChartObj.LS_SOLID);
            SimpleLinePlot thePlot2 = new SimpleLinePlot(pTransform1, Dataset1, attrib2);
            chartVu.AddChartObject(thePlot2);

            ChartAttribute attrib3 = new ChartAttribute(ChartColor.Red, 1, ChartObj.LS_SOLID);
            attrib3.SetFillColor(ChartColor.Red);
            attrib3.SetFillFlag(true);
            attrib3.SetSymbolSize(12);
            SimpleScatterPlot thePlot3 = new SimpleScatterPlot(pTransform1, Dataset3, ChartObj.CIRCLE,
attrib3);
            chartVu.AddChartObject(thePlot3);


            ChartFont legendFont = new ChartFont("Microsoft Sans Serif", 14, FontStyle.Normal);
            ChartAttribute legendAttributes = new ChartAttribute(ChartColor.Gray, 1, ChartObj.LS_SOLID,
ChartColor.LightGreen);
            // legendAttributes.SetFillFlag(false);
            legendAttributes.SetLineFlag(false);
            StandardLegend legend = new StandardLegend(0.15, 0.875, 0.9, 0.15, legendAttributes,
StandardLegend.HORIZ_DIR);
            legend.AddLegendItem("Energy Companies", ChartObj.PLUS, thePlot1, legendFont);
            legend.AddLegendItem("Software Companies", ChartObj.STAR, thePlot1, legendFont);
            legend.AddLegendItem("Predicted", ChartObj.LINE, thePlot2, legendFont);
            legend.SetLegendItemUniformTextColor(ChartColor.Black);
            chartVu.AddChartObject(legend);

            ChartFont theTitleFont = new ChartFont("Microsoft Sans Serif", 18, FontStyle.Normal,
FontWeights.ExtraBold);
            ChartTitle mainTitle = new ChartTitle(pTransform1, theTitleFont, "Theoretical vs.
Experimental Data ");
            mainTitle.SetTitleType(ChartObj.CHART_HEADER);
            mainTitle.SetTitlePosition(ChartObj.CENTER_GRAPH);

            chartVu.AddChartObject(mainTitle);


        }
    }
}
```

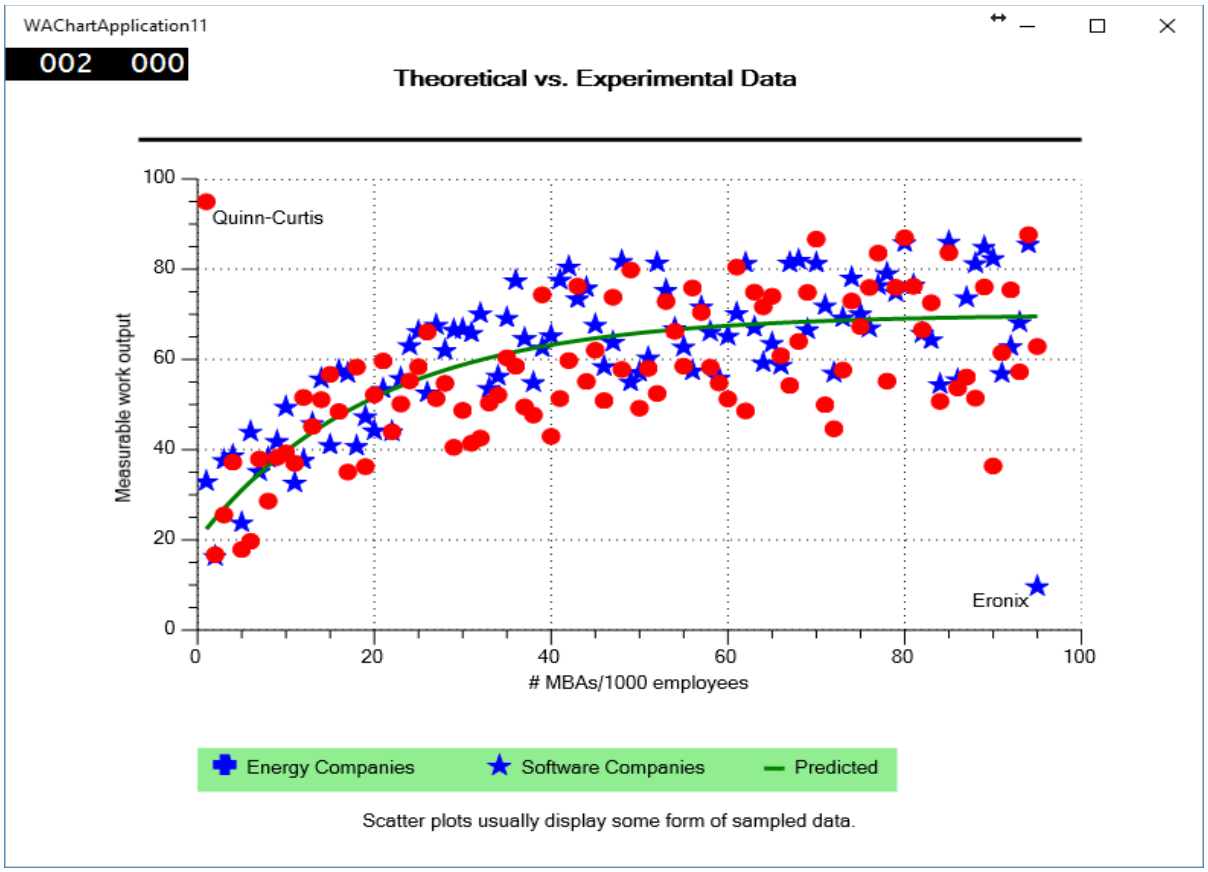
- Reference and initialize the newly created **SimpleLineAndScatterPlot** class in the MainPage.xaml.cs behind code file.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
```

```
using System.Runtime.InteropServices.WindowsRuntime;
using Windows.Foundation;
using Windows.Foundation.Collections;
using Windows.UI.Xaml;
using Windows.UI.Xaml.Controls;
using Windows.UI.Xaml.Controls.Primitives;
using Windows.UI.Xaml.Data;
using Windows.UI.Xaml.Input;
using Windows.UI.Xaml.Media;
using Windows.UI.Xaml.Navigation;
using com.quinncurtis.chart2dwa;


// The Blank Page item template is documented at http://go.microsoft.com/fwlink/?
LinkId=402352&clcid=0x409

namespace WAChartApplication11
{

    /// <summary>
    /// An empty page that can be used on its own or navigated to within a Frame.
    /// </summary>
    public sealed partial class MainPage : Page
    {

        SimpleLineAndScatterPlot slsp = null;

        public MainPage()
        {
            this.InitializeComponent();
            InitChartView();
        }

        public void InitChartView()
        {

          slsp = new SimpleLineAndScatterPlot(simplelineandscatterplot1);

          simplelineandscatterplot1.PreferredSize = new Windows.Foundation.Size(800, 600);
        }
    }
}
```

The reference to simplelineandscatterplot1.PreferredSize tells the software that the font sizes specified in the graph are with respect to a chart window of size (800, 600). If the chart is sized larger than this, the fonts will be larger, if it is sized smaller, the fonts will be smaller.

- Build the Solution (**Build | Build Solution**). If the project fails to compile you need to go back and check the errors and the previous steps. When it runs properly it the SimpleLineAndScatterPlot chart looks like:



- There are other ways to incorporate charts into your application. You can add a UserControl to your program (Add | UserControl) and place the chart entirely in the UserControl. The ChartView object is referenced in the Grid panel of the UserControl's xaml file, and initialized in the UserControl's behind code file. The UserControl derived class is then referenced in the main MainPage.xaml file. The UserControlChartExample1 demonstrates this method. The MainPage.xaml file then looks like:

```
<Page
    x:Class="UserControlChartExample1.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="using:UserControlChartExample1"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:controls="using:UserControlChartExample1"
    mc:Ignorable="d">
    <Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">
         <controls:MyUserControl1 x:Name="myusercontrol1" HorizontalAlignment="Stretch"
            Margin="10,10,10,10" VerticalAlignment="Stretch" />
    </Grid>
</Page>
```

Note that a *controls* identifier is used to point to the current namespace, and then controls:MyUserControl1 is used to specify that the chart user control is to added as a grid element.

- Or, You can just reference the ChartView class in the main MainPage.xaml file. Define the chart in the behind code file (MainPage.xaml.cs). See the MainPageChartExample example program.

Visual Basic for .Net

Initially, the Visual Studio 2015 project templates did not include a template for a Visual Basic Universal App project. We saw that as a good sign that maybe at long last Microsoft would phase out support for Visual Basic. Therefore VB examples were not included in this documentation. Now we see with a Visual Studio 2015 update, the VB Universal App templates have appeared. So we included one VB example, pretty much identical to the C# WAChartApplication1example covered in detail above. You will find it in the Quinn-Curtis\DotNet\QCChart2D\Visual Basic\examples wa folder.

If we get enough requests, we will go back and add all of the VB examples back in. Hopefully that does not happen.

26. Frequently Asked Questions

FAQs

1. Is the **QCChart2D for Windows Apps** software backward compatible with the .Net Forms version of **QCChart2D**?
2. How do you create a chart with multiple coordinate systems and axes?
3. Can I add new axes, text objects, plot objects, and images to a chart after it is already displayed; or must I create a new chart from scratch taking into account the additional objects?
4. How do you zoom charts that use multiple coordinate systems?
5. How do you select a chart object and create a dialog panel that permits editing of that objects properties?
6. How do you handle missing data points in a chart?
7. How do you update a chart in real-time?
8. How do I prevent flicker when updating my charts on real-time?
9. How do you implement drill down, or data tool tips in a chart?
10. I do not want to my graph to auto-scale. How do I setup the graph axes for a specific range?
11. How do I update my data, and auto-rescale the chart scales and axes to reflect the new data, after it has already been drawn?
12. When I use the auto-scale and auto-axis routines my semi-log chart has the logarithmic axis scaled using powers of 10 (1, 10,100, 1000, etc.) as the starting and ending values, or as the major tick interval for labeling. How do I make my log graphs start at 20 and end at 50,000, with major tick marks at 20, 200, 2000 and 20000?
13. How do I create and use custom, multi-line string labels as the axis labels for my graph?
14. How do I place more than one graph in a view?
15. How do I use your software to generate GIF files?

16. Sometimes the major tick marks of an axis are missing the associated tick mark label ?

17. How do I change the order the chart objects are drawn? For example, I want one of my grid objects to be drawn under the charts line plot objects, and another grid object to be drawn top of the charts line plot objects.

18. How to I use a scrollbar object to control horizontal scrolling of the data in my chart?

19. I am trying to plot 100,000,000 data points and it takes too long to draw the graph. What is wrong with the software and what can I do to make it faster?

20. How do I get data from my database into a chart?

21. How do I use this charting software to generate chart images "on-the-fly"?

1. Is ***the QCChart2D for Windows Apps software backward compatible with the*** *.Net Forms version of* ***QCChart2D*** ?

Yes, the QCChart2D for Windows Apps software is generally source code compatible with earlier Forms based QCChart2D for .Net. There is a comprehensive list of changes you will need to make to your source at the end of Chapter 2. You should have no problems recreating any charts that you created using our older .Net forms software.

2. How do you create a chart with multiple coordinate systems and axes?

A chart can have as many coordinate systems and axes as you want. A single coordinate system can have one or more x- and/or y-axes. The most common use for multiple axes in a single coordinate system is to place y-axes on both the left and the right sides of a chart, and x-axes above and below. The left and bottom axes usually have numeric or date labels, and the top and right axes just tick marks. This does not have to be the case though; every axis can have axis labels if you want. In general, the axis position in the chart is determined by its intercept. The default value of the intercept is set to the minimums of the coordinate system that the axis is placed in. Adjusting the intercept using the **SetAxisIntercept** method changes the position of the axis in the chart. The axis intercept value is set using units of the coordinate system at right angles to the axis. The example below, extracted from the LineFill example, places y-axes on both the left and right of the chart.

[C#]

```
TimeAxis xAxis = new TimeAxis(pTransform1);
chartVu.AddChartObject(xAxis);

TimeAxis xAxis = new TimeAxis(pTransform1);
xAxis.SetColor(Colors.White);
chartVu.AddChartObject(xAxis);

LinearAxis yAxis = new LinearAxis(pTransform1, ChartObj.Y_AXIS);
```

```
// Default places y-axis at miniumum of x-coordinate scale
yAxis.SetColor(Colors.White);
chartVu.AddChartObject(yAxis);

LinearAxis yAxis2 = new LinearAxis(pTransform1, ChartObj.Y_AXIS);
yAxis2.SetAxisIntercept(xAxis.GetAxisMax());
yAxis2.SetAxisTickDir(ChartObj.AXIS_MAX);
yAxis2.SetColor(Colors.White);
chartVu.AddChartObject(yAxis2);
```

The other common reason to have multiple axes in a chart is to delineate the simultaneous use of different coordinate systems in the chart. In this case each coordinate system has an x- and/or y-axis to differentiate it from the other coordinate systems. When the different coordinate systems are created, they usually overlay the same area of the chart. The default positioning of the axes for each coordinate system will all overlay one another, making the axes unreadable. In the y-axis case you will want to offset additional axes to the left, or to the right of the default axis position, using the **SetAxisIntecept** method. When using the **SetAxisIntercept** method, make sure you specify the position using the units of the coordinate system scale at right angles to the axis. Specify an intercept value outside of the normal scale range to offset the axes so that they do not overlap. The example below, extracted from the MultipleAxes.MultiAxesChart example, creates one x-axis, common to all of the charts because the x-scaling for all of the coordinate systems match, and five y-axes, one for each of the five different coordinate systems.

[C#]

```
CartesianCoordinates pTransform1;
CartesianCoordinates pTransform2;
CartesianCoordinates pTransform3;
CartesianCoordinates pTransform4;
CartesianCoordinates pTransform5;
.
.  // Initialize datasets, coordinate system ranges
.
// The x-scale range for pTransform1 to pTransform5 are all the same, 0 - 100
// The y-scale range for pTransform1 to pTransform5 are all different

// The plotting area for each pTransform is indentical, leaving a large open
// to the left for extra axes.
pTransform1.SetGraphBorderDiagonal(0.35, .15, .9, 0.65) ;
pTransform2.SetGraphBorderDiagonal(0.35, .15, .9, 0.65) ;
pTransform3.SetGraphBorderDiagonal(0.35, .15, .9, 0.65) ;
pTransform4.SetGraphBorderDiagonal(0.35, .15, .9, 0.65) ;
pTransform5.SetGraphBorderDiagonal(0.35, .15, .9, 0.65) ;

ChartAttribute attrib1 = new ChartAttribute (Colors.Blue, 2,ChartObj.LS_SOLID);
ChartAttribute attrib2 = new ChartAttribute (Colors.Red, 2,ChartObj.LS_SOLID);
ChartAttribute attrib3 = new ChartAttribute (Colors.Green, 2,ChartObj.LS_SOLID);
ChartAttribute attrib4 = new ChartAttribute (Colors.Orange, 2,ChartObj.LS_SOLID);
ChartAttribute attrib5 = new ChartAttribute (Colors.Magenta, 2,ChartObj.LS_SOLID);

xAxis = new LinearAxis(pTransform1, ChartObj.X_AXIS);
xAxis.SetLineWidth(2);
chartVu.AddChartObject(xAxis);

yAxis1 = new LinearAxis(pTransform1, ChartObj.Y_AXIS);
yAxis1.SetAxisIntercept(0.0);
yAxis1.SetChartObjAttributes(attrib1); // axis color matches line color
chartVu.AddChartObject(yAxis1);

yAxis2 = new LinearAxis(pTransform2, ChartObj.Y_AXIS);
yAxis2.SetAxisIntercept(-18);
```

```
yAxis2.SetChartObjAttributes(attrib2); // axis color matches line color
chartVu.AddChartObject(yAxis2);

yAxis3 = new LinearAxis(pTransform3, ChartObj.Y_AXIS);
yAxis3.SetAxisIntercept(-35);
yAxis3.SetChartObjAttributes(attrib3); // axis color matches line color
chartVu.AddChartObject(yAxis3);

yAxis4 = new LinearAxis(pTransform4, ChartObj.Y_AXIS);
yAxis4.SetAxisIntercept(-52);
yAxis4.SetChartObjAttributes(attrib4); // axis color matches line color
chartVu.AddChartObject(yAxis4);

yAxis5 = new LinearAxis(pTransform5, ChartObj.Y_AXIS);
yAxis5.SetAxisIntercept(xAxis.GetAxisMax());
yAxis5.SetAxisTickDir(ChartObj.AXIS_MAX);
yAxis5.SetChartObjAttributes(attrib5); // axis color matches line color
chartVu.AddChartObject(yAxis5);

NumericAxisLabels xAxisLab = new NumericAxisLabels(xAxis);
xAxisLab.SetTextFont(theFont);
chartVu.AddChartObject(xAxisLab);

NumericAxisLabels yAxisLab1 = new NumericAxisLabels(yAxis1);
yAxisLab1.SetTextFont(theFont);
yAxisLab1.SetAxisLabelsFormat(ChartObj.BUSINESSFORMAT);
chartVu.AddChartObject(yAxisLab1);

NumericAxisLabels yAxisLab2 = new NumericAxisLabels(yAxis2);
yAxisLab2.SetTextFont(theFont);
chartVu.AddChartObject(yAxisLab2);

NumericAxisLabels yAxisLab3 = new NumericAxisLabels(yAxis3);
yAxisLab3.SetTextFont(theFont);
chartVu.AddChartObject(yAxisLab3);

NumericAxisLabels yAxisLab4 = new NumericAxisLabels(yAxis4);
yAxisLab4.SetTextFont(theFont);
chartVu.AddChartObject(yAxisLab4);

NumericAxisLabels yAxisLab5 = new NumericAxisLabels(yAxis5);
yAxisLab5.SetTextFont(theFont);
chartVu.AddChartObject(yAxisLab5);

ChartFont axisTitleFont = new ChartFont("Microsoft Sans Serif", 10, FontStyle.Normal,
FontWeights.Bold);
AxisTitle xaxistitle = new AxisTitle( xAxis, axisTitleFont, "Event Partition");
chartVu.AddChartObject(xaxistitle);

ChartGrid xgrid = new ChartGrid(xAxis, yAxis1,ChartObj.X_AXIS, ChartObj.GRID_MAJOR);
chartVu.AddChartObject(xgrid);

SimpleLinePlot thePlot1 = new SimpleLinePlot(pTransform1, Dataset1, attrib1);
chartVu.AddChartObject(thePlot1);

SimpleLinePlot thePlot2 = new SimpleLinePlot(pTransform2, Dataset2, attrib2);
chartVu.AddChartObject(thePlot2);

SimpleLinePlot thePlot3 = new SimpleLinePlot(pTransform3, Dataset3, attrib3);
chartVu.AddChartObject(thePlot3);

SimpleLinePlot thePlot4 = new SimpleLinePlot(pTransform4, Dataset4, attrib4);
chartVu.AddChartObject(thePlot4);

SimpleLinePlot thePlot5 = new SimpleLinePlot(pTransform5, Dataset5, attrib5);
chartVu.AddChartObject(thePlot5);
```

3. **Can I add new axes, text objects, plot objects, and images to a chart after it is already displayed; or must I create a new chart from scratch taking into account the additional objects?**

There are two ways to add new objects to a chart. The first way is to create all objects when the chart is initially created, but disable the ones that you do not want to show up when the chart is initially rendered. Enable the objects when you want them to show up. Use the chart objects **SetChartObjEnable** method to enable/disable the object. This is useful if you are creating an animated chart where you want the chart to sequence through a predefined series of steps. The second way you add new chart objects to the **ChartView** using the **ChartView.AddChartObject** method. In both cases you need to call the **ChartView.UpdateDraw()** method after any changes are made.

The example below, extracted from the **CustomChartDataCursor** class, creates a new **Marker** object and **NumericLabel** object each time a mouse button clicked.

[C#]

```
Marker amarker = new  Marker(GetChartObjScale(), MARKER_BOX,
           nearestPoint.GetX(), nearestPoint.GetY(),  10.0, PHYS_POS);
chartVu.AddChartObject(amarker);
rNumericLabelCntr += 1.0;
// Add a numeric label the identifies the marker
pointLabel = new NumericLabel(GetChartObjScale(),
           textCoordsFont, rNumericLabelCntr, nearestPoint.GetX(),
           nearestPoint.GetY(), PHYS_POS, DECIMALFORMAT, 0);
// Nudge text to the right and up so that it does not write over marker
pointLabel.SetTextNudge(5,-5);
chartVu.AddChartObject(pointLabel);
chartVu.UpdateDraw();
```

4. **How do you zoom charts that use multiple coordinate systems?**

The **ChartZoom** class will zoom one or more simultaneous coordinate systems. The example program SuperZoom zooms a chart that has one x-axis and five y-axes. Use the **ChartZoom** constructor that accepts an array of coordinate system objects.

5. **How do you select a chart object and create a dialog panel that permits editing of that objects properties?**

The **QCChart2D for Windows Apps** library does not include predefined dialogs for editing chart object properties. The look, feel and details of such dialogs are application specific and it is up to the application programmer to provide these. The property editor tables common to many packages are designed to be used by developers, not end users.

You can add your own dialogs that edit the characteristics important to your end users. If you want to select the chart object by pressing a mouse button while the cursor is on the object, use the **FindObj** class. Override the **OnMouseDown** method and invoke the appropriate

dialog panel there. The following example is extracted from the **EditChartExample** example program.

[C#]

```
public class SimpleLineAndScatterPlot
  {
      ChartView gWG;
      LinearAxis xAxis = null;
      LinearAxis yAxis = null;
      SimpleLinePlot thePlot2 = null;
      SimpleScatterPlot thePlot1 = null;
      SimpleScatterPlot thePlot3 = null;

      class CustomFindObj : FindObj
      {
          SimpleLineAndScatterPlot charttObj = null;
          public CustomFindObj(SimpleLineAndScatterPlot component)
              : base(component.gWG)
          {
              charttObj = component;
          }

          public void InvokeLineDialog(GraphObj graphobj)
          {
              charttObj.LineEdit_Click(graphobj);
          }
          public override void OnMouseDown(PointerRoutedEventArgs mouseevent)
          {
              base.OnMouseDown(mouseevent);
              GraphObj selectedObj = GetSelectedObject();
              if (selectedObj != null)
              {

                  // Check for a specific object
                  if ((selectedObj == charttObj.xAxis) ||
                      (selectedObj == charttObj.yAxis) ||
                      // or check for for all classes inheriting from a specific type

                      (ChartSupport.IsKindOf(selectedObj, "SimplePlot")))
                       InvokeLineDialog(selectedObj);
              }
          }
      }

      public async void  LineEdit_Click(GraphObj graphobj)
      {
          EditLineDialog editlinedialog = new EditLineDialog()
          {
              CurrentLineWidth = (int)graphobj.ChartObjAttributes.LineWidth,
             CurrentColor = graphobj.ChartObjAttributes.PrimaryColor
          };

          ContentDialogResult dlgresult = await editlinedialog.ShowAsync();
          if (dlgresult == ContentDialogResult.Primary)
          {
              // line width
              graphobj.ChartObjAttributes.LineWidth = editlinedialog.CurrentLineWidth;
              // line color
              graphobj.ChartObjAttributes.PrimaryColor = editlinedialog.CurrentColor;
              // Fill symbol if a scatter plot type
              if (ChartSupport.IsKindOf(graphobj, "SimpleScatterPlot"))
                  graphobj.ChartObjAttributes.FillColor = editlinedialog.CurrentColor;

              gWG.UpdateDraw();
          }

      }
```

The **EditLineDialog class** need to be written by the programmer. Sample classes are found in the EditChartExample example. This example reveals two strange weaknesses of UWP; there is no common control color picker dialog, and no common font chooser dialog. We use a simple scroll list box for to choose a color.

6. How do you handle missing data points in a chart?

There are two ways to handle missing, or bad data. The first is to mark the data point in the dataset invalid, using the datasets **SetValidData** method. The second is to set the x- and/or y-value of the bad data point to the designated bad data value, **ChartObj.rBadDataValue**. Currently this value is set equal to the value of System.Double.MaxValue. Either method will prevent the data point from being displayed in a chart. If the bad data value is part of a line plot, a gap will appear in the line plot at that point. Bad data points are not deleted from a dataset.

7. How do you update a chart in real-time?

In general, real-time updates involve adding new objects to a chart, or modifying existing objects that are already in the chart. Once the object is added or changed, call the **ChartView.UpdateDraw()** method to force the chart to update using the new values. Objects can be added or modified based on some external event, or in response to a timer event created using **DispatcherTimer**. Make all changes for a given event and call the **ChartView.UpdateDraw** method once. The position of most **GraphObj** derived objects is set or modified using one of the objects **SetLocation** methods. New data points can be added to an existing dataset using one of the datasets **AddDataPoint**, **AddTimeDataPoint**, **AddGroupDataPoints** or **AddTimeGroupDataPoints** methods. **ChartPlot** derived objects that use datasets will update to reflect the new values when the **ChartView.UpdateDraw** method is called. If the coordinates of the new data points are outside of the x- and y-limits of the current coordinate system it may be necessary to rescale the coordinate system so that the new points show up; otherwise the new data points will be clipped. The new scale values can be set explicitly, or calculated using one of the auto-scale methods. The example program DynamicCharts demonstrates various ways to update charts in real-time.

If you want to change points in an existing dataset, but not the size of the dataset, call the datasets appropriate **SetXDataValue**, **SetYDataValue**, or **SetDataPoint** methods. The dataset has its own copy of the data so you must change these values, not the original values you used to initialize the dataset. If you plan to change every value in the dataset, you can do that point by point, or create a new dataset and swap that in for the old dataset using the plot objects **SetDataset** or **SetGroupDataset** method. Call the **ChartView.UpdateDraw** method to force the chart to update using the new values.

8. How do I prevent flicker when updating my charts on real-time?

In general, the retained graphics system of UWP is always doubled buffered does not produce flicker.

9. How do you implement drill down, or data tool tips in a chart?

Implementing drill down or tool tips consists of three major parts:

- Trapping a mouse event and determining the mouse cursor position in device and physical coordinates.
- Identifying the chart object that intersects the mouse event.
- Displaying appropriate information about the chart object.

There are many classes that aid in one or more of these functions. The **MouseListener** class will trap a mouse event in the chart view. The **FindObj** class will filter and return the chart object, if any, that intersects the mouse cursor when a mouse button is pressed. The **MoveObj** class will filter, select and move a chart object as the mouse is dragged across the chart. The **DataToolTip** class will find the data point in a chart nearest the mouse cursor and display xy information about the data point as a popup **ChartText** display. The **DataToolTip** can also be customized for the display of custom information about the selected data point. It only takes a few lines to add a simple y-value tool tip to an existing chart.

[C#]

```
DataToolTip datatooltip = new DataToolTip(chartVu);
datatooltip.SetEnable(true);
chartVu.SetCurrentMouseListener(datatooltip);
```

Most all of the example programs have charts which use data tool tips. See the SimpleLinePlots examples: LineFill, LinePlotSegments, and SimpleLineAndScatterPlot. Also see the Bargraphs examples: SimpleBars, DoubleBarPlot, HistogramBars, GroupBargraphs, and FloatingBars.

10. I do not want to my graph to auto-scale. How do I setup the graph axes for a specific range?

Auto-scaling has two parts. The first is the auto-scaling of the coordinate system based on one or more datasets. The second part is the auto-scaling of the axes that reside in the coordinate system. Manually scale the coordinate system and axes by calling the appropriate constructors. For example:

[C#]

```
ChartCalendar xMin = new ChartCalendar(1996, ChartObj.FEBRUARY,  5);
ChartCalendar xMax = new ChartCalendar(2002, ChartObj.JANUARY,  5);
double yMin = 0;
double yMax =  105;

TimeCoordinates simpleTimeScale;
simpleTimeScale = new TimeCoordinates(xMin, yMin, xMax, yMax);
// Create the time axis (x-axis is assumed)
TimeAxis xAxis = new TimeAxis(simpleTimeScale);
// Create the linear y-axis
LinearAxis yAxis = new LinearAxis(simpleTimeScale, ChartObj.Y_AXIS);

// Create the ChartView object to place graph objects in.
ChartView chartVu  = new ChartView();

// Add the x- and y-axes to the chartVu object
chartVu.AddChartObject(xAxis);
chartVu.AddChartObject(yAxis);
```

The documentation for the various coordinate system and axis classes includes examples of manual scaling.

11. How do I update my data, and auto-rescale the chart scales and axes to reflect the new data, after it has already been drawn?

Updating data was discussed in FAQ # 6. If you want the chart to rescale based on the new data, call the appropriate coordinate systems auto-scale method, followed by the auto-axis methods of all related axes. Then call the **ChartView.UpdateDraw** method. For example:

[C#]

```
// Create the ChartView object to place graph objects in.
TimeSimpleDataset Dataset1 =  new TimeSimpleDataset("Sales",x1,y1);

TimeCoordinates simpleTimeCoordinates = new TimeCoordinates();
simpleTimeCoordinates.AutoScale(Dataset1,
       ChartObj.AUTOAXES_FAR , ChartObj.AUTOAXES_FAR);
ChartView chartVu  = new ChartView();
// Create the time axis (x-axis is assumed)
TimeAxis xAxis = new TimeAxis(simpleTimeCoordinates);
// Create the linear y-axis
LinearAxis yAxis = new LinearAxis( simpleTimeCoordinates, ChartObj.Y_AXIS);
.
.
.
// The following code would be in the code handling the rescale event
// Rescale chart based on a modified Dataset1 datset
simpleTimeCoordinates.AutoScale(Dataset1,
                ChartObj.AUTOAXES_FAR , ChartObj.AUTOAXES_FAR);
xAxis.CalcAutoAxis();
yAxis.CalcAutoAxis();
// Redraw the chart using the rescaled coordinate system and axes
chartVu.UpdateDraw();
```

12. When I use the auto-scale and auto-axis routines my semi-log chart has the log axis scaled using powers of 10 (1, 10,100, 1000, etc.) as the starting and ending values, or as the major tick interval for labeling. How do I make my log graphs start at 20 and end at 50,000, with major tick marks at 20, 200, 2000 and 20000?

The auto-scale routines for logarithmic coordinate systems will always select a power of 10 for the minimum and maximum value of the scale. You can use the auto-scale routine and then override the minimum and/or maximum values for the logarithmic scale. The default **LogAxis** constructor will pick up on the minimum of the coordinate system and use that as the axis tick mark origin. Or you can leave the coordinate system unchanged, and change the starting point of the axis tick marks using the axis **SetAxisTickOrigin** method. The example below is derived from the Logarithmic example code.

[C#]

```
GroupDataset Dataset1 =  new GroupDataset("First",x1,y1);

CartesianCoordinates pTransform1 =  new CartesianCoordinates(ChartObj.LOG_SCALE,
                                    ChartObj.LINEAR_SCALE);
pTransform1.AutoScale(Dataset1, ChartObj.AUTOAXES_FAR,  ChartObj.AUTOAXES_FAR);

pTransform1.SetScaleStartX(20); // Force start of scale at 20, AutoScale will
                                // always choose a power of 10 decade.
LogAxis xAxis = new LogAxis(pTransform1, ChartObj.X_AXIS);
xAxis.SetAxisTickOrigin(20);
chartVu.AddChartObject(xAxis);
```

13. How do I create and use custom, multi-line string labels as the axis labels for my graph?

The **StringAxisLabels** class should be used to create multi-line axis labels. Insert the "\n" new line character to add additional lines to each string used to define the string axis labels. The example below is from the AxisLabels example program.

[C#]

```
String []xstringlabels =
             {
               "",
               "Western"+"\n"+"Sales"+"\n"+"Region",
               "Eastern"+"\n"+"Sales"+"\n"+"Region",
               "Southern"+"\n"+"Sales"+"\n"+"Region",
               "Northern"+"\n"+"Sales"+"\n"+"Region"};

StringAxisLabels xAxisLab5 = new StringAxisLabels(xAxis5);
xAxisLab5.SetAxisLabelsStrings(xstringlabels,5);
xAxisLab5.SetTextFont(graph5Font);
chartVu.AddChartObject(xAxisLab5);
```

14. How do I place more than one graph in a view?

One way to create multiple charts is to create multiple instances of the **ChartView** class and add each **ChartView** object to a container object such as a Window or **UserControl**. The UWP layout manager is used to manage the position and size of each **ChartView**. Another way is to place multiple charts in the same **ChartView** object. This makes it easier to guarantee alignment between the axes of separate graphs. The trick to doing this is to create separate coordinate system objects (**CartesianCoordinates**, **TimeCoordinates** or **PolarCoordinates**) for each chart, and to position the plot area of each coordinate system so that they do not overlap. Use one of the coordinate systems **SetGraphBorder**… methods. Many of the examples use this technique, including Bargraphs.GroupBargraphs, Bargraphs.DoubleBarPlot, FinancialExamples.OHLCChart, FinancialExamples.FinOptions, DynamicCharts.DynPieChart, PieCharts.PieAndLineChart and PieCharts.PieAndBarChart. The example below was extracted from the FinancialExamples.OHLCChart class.

[C#]

```
pTransform1 = new TimeCoordinates();
pTransform1.SetGraphBorderDiagonal(0.1, .15, .90, 0.6) ;

pTransform2 = new TimeCoordinates();
pTransform2.SetGraphBorderDiagonal(0.1, .7, .90, 0.875) ;
```

15. How do I use your software to generate GIF files?

Unlike the JPEG image file format, the GIF file format uses a proprietary data compression algorithm known as LZW. The patent on the LZW compression algorithm is owned by the large computer/data processing company Unisys. Programmers who write commercial applications that use this file format may be subject to paying Unisys royalties. The BufferedImage class will out GIF files using the standard .Net image encoder.

According to Wikipedia, all of the patents associated with GIF file format have expired, and you are free to use that file format in your work.

16. Sometimes the major tick marks of an axis are missing the associated tick mark label ?

The axis labeling routines are quite intelligent. Before the label is drawn at its calculated position, the software does a check to see if the bounding box of the new axis label intersects the bounding box of the previous axis label. If the new label is going to overlap the previous label, the label is skipped. You can override this default behavior by calling the objects **SetOverlapLabelMode** method.

```
SetOverlapLabelMode (ChartObj.OVERLAP_LABEL_DRAW);
```

Another option, for horizontal axes only, is to stagger the tick mark labels. A stagger automatically alternates the line on which the tick mark label is placed.

```
SetOverlapLabelMode (ChartObj.OVERLAP_LABEL_STAGGER);
```

17. How do I change the order the chart objects are drawn? For example, I want one of my grid objects to be drawn under the charts line plot objects, and another grid object to be drawn top of the charts line plot objects.

There are two ordering methods used to render chart objects. The first method renders the objects in order, as added to the **ChartView** object. Objects added to the view last are drawn on top of objects added first. The second method renders the objects according to their z-order. Objects with the lowest z-order values are rendered first. Objects with equal z-order values are rendered in the ordered they are added to the **ChartView** object. The second method (z-order rendering) is the default method of object rendering used by the **ChartView** class. This default behavior can be changed by call the **ChartView.SetZOrderSortEnable(false**) method.

You can change the default z-order value on an object-by-object basis. Call the **GraphObj.SetZOrder** method to change the z-order for any given object.

See the section in the manual titled ***Rendering Order of GraphObj Objects*** for information about the default z-values for all chart objects

The example below sets the z-order value of grid1 to something less than the default value (50) of **ChartPlot** objects, and the z-order value of grid2 to something greater than the default value.

[C#]

```
ChartView chartVu = new ChartView();
.
.
.

ChartGrid grid1 =  new ChartGrid(xAxis, yAxis, ChartObj.Y_AXIS, ChartObj.GRID_MAJOR);
grid1.SetZOrder(40); // This is actually the default value for the grid z-order
chartVu.AddChartObject(grid1);

ChartGrid grid2 =  new ChartGrid(xAxis, yAxis, ChartObj.Y_AXIS, ChartObj.GRID_MINOR);
grid2.SetZOder(150); // ChartGrid is drawn after ChartPlot objects
                     // which have default z-value of 50
chartVu.AddChartObject(grid2);
```

18. How to I use a ScrollBar object to control horizontal scrolling of the data in my chart?

The ChartView class has two built-in scroll bars you can use in your program to control chart scrolling. The example program FormControlExamples.LinePlotScrollBar uses two scroll bars, a horizontal scroll bar to control scrolling of the x-axis, and a vertical scroll bar that

controls the magnitude of the y-axis. You need to enable the HScrollBar1 and VScrollBar1 scrollbars in the ChartView. Hook up the the hScrollBar1_Scroll and vScrollBar1_Scroll event listeners to the Scroll event of each scroll bar.

[C#]

```
public void UpdateXScaleAndAxes(int index)
{
    int startindex = index;
    pTransform1.SetScaleStartX((double)startindex);
    pTransform1.SetScaleStopX((double)(startindex + 100));
    xAxis.CalcAutoAxis();
    yAxis.CalcAutoAxis();
    xAxisLab.CalcAutoAxisLabels();
    yAxisLab.CalcAutoAxisLabels();
    chartVu.UpdateDraw();
}

public void UpdateYScaleAndAxes(int index)
{
    int startindex = index;
    pTransform1.SetScaleStartY((double)-startindex);
    pTransform1.SetScaleStopY((double)startindex);
    xAxis.CalcAutoAxis();
    yAxis.CalcAutoAxis();
    xAxisLab.CalcAutoAxisLabels();
    yAxisLab.CalcAutoAxisLabels();
    chartVu.UpdateDraw();
}


private void InitializeChart(ChartView chartvu)
{
    int i;

    chartVu = chartvu;

    .
   .
   .

    chartVu.EnableHScrollBar1 = true;
    chartVu.EnableVScrollBar1 = true;

    chartVu.HScrollBar1.Minimum = 0;
    chartVu.HScrollBar1.Maximum = 200;
    chartVu.HScrollBar1.Value = 0;
    chartVu.HScrollBar1.SmallChange = 2;
    chartVu.HScrollBar1.LargeChange = 20;

    chartVu.VScrollBar1.Minimum = 0;
    chartVu.VScrollBar1.Maximum = 100;
    chartVu.VScrollBar1.Value = 50;
    chartVu.VScrollBar1.SmallChange = 1;
    chartVu.VScrollBar1.LargeChange = 5;

    chartVu.HScrollBar1.ValueChanged += hScrollBar1_Scroll;
    chartVu.VScrollBar1.ValueChanged += vScrollBar1_Scroll;

    UpdateYScaleAndAxes((int)chartVu.VScrollBar1.Value);
    UpdateXScaleAndAxes((int)chartVu.HScrollBar1.Value);

}
```

```
internal void hScrollBar1_Scroll(object sender, object e)
{
   RangeBaseValueChangedEventArgs ex = (RangeBaseValueChangedEventArgs)e;
    UpdateXScaleAndAxes((int)ex.NewValue);
}

internal void vScrollBar1_Scroll(object sender, object e)
{
  RangeBaseValueChangedEventArgs ex = (RangeBaseValueChangedEventArgs)e;
  UpdateYScaleAndAxes((int)ex.NewValue);
}
```

There are many other examples of Form components interacting with charts. The ContourPlots.ContourLinePlot example program uses a **CheckBox** object to specify which contours are to be displayed. In the MultipleAxes.OHLCChart example a **Scrollbar** controls the time axis of a stock market OHLC chart. The MultiAxes.MultiAxesChart example uses **Button** objects to select the x-axis range.

19. I am trying to plot 100,000,000 data points and it takes too long to draw the graph. What is wrong with the software and what can I do to make it faster?

The software runs as fast as we can make it. We do not have any hidden switches that will speed up the software. What you need to do is to step back and think about the best way to display your data.

A fundamental issue that many programmers fail to consider is the relationship between the resolution of the rasterized screen image of the plot and the resolution of the data. A typical chart image will have 500-1000 pixels as the horizontal resolution of the plotting area. This would imply that in the 100M data point example above, every horizontal pixel would represent 50K to 100K data points. Obviously this is a terrible mismatch. In fact it is a bad match for datasets that have more than a couple of thousands points.

So what you do is compress the data before it is displayed. Take the 100M data points and compress them down to 2K data points. The data compression can take several forms. You can take an average of every N points. The resulting dataset will be reduced by a factor of N. You can also find the sum for every N points, the minimum value of every N points, the maximum of every N points, or both the minimum and maximum of every 2N points. The last compression method, minimum and maximum, will always capture any minimums and maximum in the data. The result is that a 2000 point compressed dataset, where there are at least two data points per pixel of horizontal resolution, will look just like the 100,000,000 point dataset, only display hundreds of times faster. The **Dataset** classes all include compression methods (**SimpleDataset.CompressSimpleDataset**, **GroupDataset.CompressGroupDataset**, **TimeSimpleDataset.CompressTimeSimpleDataset** and **TimeGroupDataset.CompressTimeGroupDataset, TimeGroupDataset.CompressTimeFieldSimpleDataset,**

TimeGroupDataset.CompressTimeFieldGroupDataset) that operate on the existing dataset and return a new, compressed dataset. The **CompressTimeFieldSimpleData** and **CompressTimeFieldGroupDataset** are particular useful because they do not use a fixed sample size of N, instead they compress data so that adjacent time values are an increment of a specific time field (ChartObj.DAY_OF_YEAR, ChartObj.WEEK_OF_YEAR, ChartObj.MONTH, ChartObj.Year). Compressing data by month and year obviously requires a varying sample size.

Once created, connect the compressed dataset to the **ChartPlot** object used to display the dataset.

[C#]

```
nNumPnts = 1000000;
TimeSimpleDataset RawDataset = new
      TimeSimpleDataset("Raw", xtimedata,  ydata,nNumPnts);
int compressXmode = ChartObj.DATACOMRESS_AVERAGE;
int compressYmode = ChartObj.DATACOMRESS_MINMAX;
int compressTimeField = Calendar.MONTH;
TimeSimpleDataset CompressedDataset =
       RawDataset.CompressTimeFileSimpleData( compressXmode,
                                              compressYmode,
                                              compressTimeField,
                                              0, nNumPnts,"Compressed");
```

20. How do I get data from my database into a chart?

The real question is: How do you get data from your database into a simple .Net program, storing sequential data values in data array variables. This is up to you and is independent of the charting software. We recommend that you use the SQL database classes that are part of .Net and study the documentation provide by Microsoft and other sources, such as the O'Reilly programming books. Once you can read individual data elements of your data base it is a trivial matter to place the numeric and calendar data into simple .Net array variables and from there plot the data.

21. How do I use this charting software to generate chart images "on-the-fly" from a server?

The **BufferedImage** class creates chart images independent of any physical display context. The BufferedImage class uses standard .Net encoders to generate image files. You can use this image as an image object in an HTML page.

INDEX