Creating Chart Styles for Wealth-Lab Pro®

Introduction

This document explains how to create new **Chart Styles** for Wealth-Lab Pro®. In Wealth-Lab Pro, all **Chart Styles** are classes that derive from the **ChartStyle** abstract base class. This applies to the standard chart styles that ship with the product, such as Line, Candlestick, and Bar Chart Styles. If you want to provide a custom settings interface for your Chart Style, your class can also implement the **ICustomSettings** interface, described below.

To build your own Chart Style, the process is simple:

1. Create a **Class Library** project in Visual Studio that will contain one or more Chart Styles.
2. Add a reference to the **WealthLab.dll** assembly in your project's References section. This assembly contains classes in the WealthLab namespace.
3. Create a class (or classes) that derive from the **ChartStyle** base class. **ChartStyle** is defined in the WealthLab namespace.

To test your Chart Style, do the following:

- Drop your **Chart Style Assembly** (created above) into the Wealth-Lab Pro installation folder.
- Execute Wealth-Lab Pro. Your Chart Style(s) should be visible and available.

ChartStyle Base Class

Your Chart Style must override some abstract methods defined in the **ChartStyle** base class. The **ChartStyle** base class also contains a number of properties and methods that will find useful when initializing the data for your chart style, and rendering the chart.

Method Overrides

You need to override the following abstract **ChartStyle** methods in your derived class.

public override void Initialize()

Wealth-Lab Pro calls this method whenever the symbol changes on a chart. Here you can build any internal data structures that you need. Access the **Bars** property that is provided in **ChartStyle** to obtain the bar data that is being charted. If you are implementing a simple Chart Style, it is quite likely that you will not need to do anything here. But if your Chart Style is more complex, you may need to create internal data structures that are based on the underlying open, high, low, close and volume data available in the **Bars** property.

protected override void InitializeBarWidths()

Wealth-Lab Pro calls this method whenever the width of the individual bars need to be calculated. See the section below on Variable Bar Widths for more information.

public override void RenderBars(Graphics g)

This is the meat of your Chart Style. Wealth-Lab Pro calls this whenever the chart bars need to be rendered. You will render the bars of the chart on the **Graphics** object provided as a parameter. The bar data comes from the **Bars** object that is also passed to you. You'll use the **IChartHost** interface to map bar/price coordinates to pixel x/y coordinates on the **Graphics** object. The section below goes into more detail on rendering the bars.

public override string FriendlyName In the property get, return a "friendly" name that describes the Chart Style in a compact form. Wealth-Lab Pro will use your friendly name as the caption for menu items or other controls that need to represent your Chart Style.
public override Bitmap Glyph In the property get, you should return a 16x16 Image that Wealth-Lab will use when it needs to represent your Chart Style on a button or menu control. Use **Fuchsia** as the transparent color for your image.

Variable Bar Width Chart Styles

Most Chart Styles, including the common Line, Candlestick, and Bar, render bars that are all the same width. Other, more complex Chart Styles, like Equivolume, Kagi, and Point & Figure, use variable width bars, where each bar can have a different width, or certain bars are not rendered on the chart at all. If your Chart Style will employ variable width bars, you need to perform some special processing in the **InitializeBarWidths** method.

You need to tell Wealth-Lab the width of each individual bar in your implementation of **InitializeBarWidths**. Accomplish this by looping through all of the bars of the **Bars** property that the **ChartStyle** base class exposes to you. You can examine the open, high, low, close and volume values to determine how wide a bar needs to be.

When calculating the width of each bar, you should also consider the default bar spacing that the user has established on their chart. Your variable-width bars might take the chart's bar spacing into account when determining individual bar width. Obtain the current bar spacing from the **BarSpacing** property that is also provided by the **ChartStyle** base class.

For each bar of the chart, once you calculate that bar's width, call **SetBarWidth** to communicate this value back to Wealth-Lab. Some Chart Styles suppress certain bars, and group them into a single bar for rendering. If certain bars should not be drawn, call **SetBarWidth** for these bars and pass zero as the width parameter.

Rendering the Bars

The **RenderBars** method is called whenever the chart needs to be drawn. You only need to draw the bars of the **Bars** property that are visible on the chart. Use the properties **RightEdgeBar** and **LeftEdgeBar** to determine the range of bars that needs to be drawn. Typically, you will use a for loop such as the one below as the main drawing loop:

```
for (int bar = LeftEdgeBar; bar <= RightEdgeBar; bar++)
{
   //Render this bar
}
```

To determine where to draw a bar, you will need to convert the bar number to a horizontal (x) pixel coordinate, and convert price values to vertical (y) pixel coordinates. Use **ConvertBarToX** to convert a bar number of an x pixel coordinate.

Converting price values to y coordinates is just a bit more complicated. You first need the pane that the bar is being drawn on, this is always the price pane. Use the **PricePane** property to get an

instance of a **ChartPane** class that represents the price pane. You can then call the **ChartPane's ConvertValueToY** method to convert a price value to a vertical (y) pixel coordinate.

For each bar that you render, you should call **GetBarColor** to determine the color of the bar. The color of the bar is normally based on whether the bar is considered an "up" bar or a "down" bar; if the bar closes higher than it's open it is considered "up". The underlying **ChartRenderer** component can associate two different colors for these values. But individual bars may be assigned their own colors in a variety of ways, so it is important to use **GetBarColor** to get the correct color of each bar that you draw.

Rendering the Ghost Bar

When the **ShouldDrawGhostBar** property is true, you can optionally render the ghost bar. Access the horizontal (x) pixel position of the ghost bar from the **GhostBarXPosition** property. To get at the partial ghost bar values of the Bars object, use Open.PartialValue, High.PartialValue, etc. Use **GhostBarColor** to access the color to use for the ghost bar.

ChartStyle Properties and Methods

The **ChartStyle** base class contains a number of properties and methods that are useful when initializing and rendering Chart Styles. Since your class derives from **ChartStyle**, it automatically has access to all of this functionality.

public Bars Bars Provides access to the **Bars** object that is being charted. The **Bars** object contains a **Count** property, and one **DataSeries** each for the chart's Open, High, Low, Close, and Volume values. To get at the partial Ghost Bar values of the Bars object, use Open.PartialValue, High.PartialValue, etc.
public int LeftEdgeBar **public int** RightEdgeBar These properties return the leftmost and rightmost bars that should be rendered on the chart. Bar numbers are indexed starting at zero and ending at **Bars**.**Count** - 1.
public int ConvertBarToX(**int** bar) Returns the horizontal (x) pixel coordinate for the specified bar number. If this bar is not in the current visible region of the chart, this method returns -1.
public int ConvertXToBar(**int** x) Converts a horizontal (x) pixel coordinate to the closest matching bar number. If the specified pixel coordinate does not map to a valid bar number, the method returns -1.
public bool ShouldDrawGhostBar Returns true if partial bar data exist and the ghost bar should be rendered, otherwise false.
public int GhostBarXPosition Returns the horizontal (x) pixel coordinate for the ghost bar.
public Color BackgroundColor This property returns the background color used for the chart.
public Color GetBarColor(**int** bar)

Returns the color that should be used to render an individual bar.

public Color GhostBarColor

Returns the color that should be used to render the ghost bar.

public int BarSpacing

This property returns the current bar spacing that is established for the chart. Use this value when initializing bar widths if your Chart Style supports variable-width bars.

public int GetBarWidth(**int** bar)

Returns the width of an individual bar.

public void SetBarWidth(**int** bar, **int** value)

Sets the width of an individual bar. Only call this method if your Chart Style supports variable width bars. And if it does, you should call **SetBarWidth** for each bar in the supplied **Bars** object in the **InitializeBarWidths** method.

public ChartPane PricePane

This property returns the instance of the price pane of the chart. Use the **ChartPane.ValueToY** method to convert price values to vertical (y) pixel coordinates.

public ChartPane VolumePane

This property returns the instance of the volume pane of the chart. Use this instance's **ChartPane.ValueToY** method to convert volume values to vertical (y) pixel coordinates.

public IList<ChartPane> Panes

This property returns a list of **ChartPane** objects that represent all of the panes on the chart.

public int ChartWidth

This property returns the width of the chart being drawn, in pixels.

public int ChartHeight

This property returns the height of the chart being drawn, in pixels.

ICustomSettings Interface

Your Chart Style class can optionally implement the **ICustomSettings** interface to support user-configurable custom settings. You must then define a **UserControl**-derived class that contains the user interface for the Chart Style's settings. Wealth-Lab obtains your **UserControl** and communicates about the settings using the **ICustomSettings** interface.

- ICustomSettings documentation

When defining keys for your custom settings values, prefix the key name with the class name of the ChartStyle, to avoid name collisions in the custom settings file. See the example below.

Chart Style Example

The example code below renders a line based chart. For more complex styles you will typically need to access the Bar's Open, High, and Low values in additional to Close.

```
/*
    LineChartStyle
    Implements the Line chart ChartStyle, simply draw a line from close to close
*/

using System;
using System.Collections.Generic;
using System.Text;
using System.Drawing;
using WealthLab;
using Fidelity.Components;

namespace WealthLab.ChartStyles
{
    public class LineChartStyle : ChartStyle, ICustomSettings
    {
        //---*** Methods to override ***---

        //Nothing to do in Initialize
        public override void Initialize()
        {
        }

        //No initialization needed, fixed bar width chart style
        protected override void InitializeBarWidths()
        {
        }

        //Render line chart
        public override void RenderBars(Graphics g)
        {
            //Set initial position of line
            int x = ConvertBarToX(LeftEdgeBar);
            int y = PricePane.ConvertValueToY(Bars.Close[LeftEdgeBar]);
            int x2, y2;

            //Create a round-capped pen for drawing the line chart
            Pen PenLine = new Pen(Color.Black);
            PenLine.Width = _lineWidth;
            PenLine.StartCap = System.Drawing.Drawing2D.LineCap.Round;
            PenLine.EndCap = PenLine.StartCap;

            //Main bar rendering loop
            for (int bar = LeftEdgeBar + 1; bar <= RightEdgeBar; bar++)
            {
                //Determine color to draw line
                PenLine.Color = GetBarColor(bar);

                //Get the destination point of the line
                x2 = ConvertBarToX(bar);
                y2 = PricePane.ConvertValueToY(Bars.Close[bar]);

                //Draw the line
                g.DrawLine(PenLine, x, y, x2, y2);

                //destination point becomes source point of next line
                x = x2;
                y = y2;
            }
            //Draw line to ghost bar (if displayed)
            if (ShouldDrawGhostBar)
            {
```

```
            x2 = GhostBarXPosition;
            y2 = PricePane.ConvertValueToY(Bars.Close.PartialValue);
            PenLine.Color = GhostBarColor;
            g.DrawLine(PenLine, x, y, x2, y2);
         }

         //Dispose of the Pen resource
         PenLine.Dispose();
      }

      //Return the friendly name of this ChartStyle
      public override string FriendlyName
      {
         get
         {
            return "Line";
         }
      }

      //Get 16x16 image
      public override Bitmap Glyph
      {
         get
         {
            return Properties.Resources.LineChartStyle;
         }
      }

      //---*** ICustomSettings implementation ***---

      //Changes to the UI have been accepted
      public void ChangeSettings(System.Windows.Forms.UserControl ui)
      {
         _lineWidth = (int)_settingsUI.numWidth.Value;
      }

      //Read default settings from host
      public void ReadSettings(ISettingsHost host)
      {
         _lineWidth = host.Get("LineChartStyle.LineWidth", 1);
      }

      //Return the settings UI
      public System.Windows.Forms.UserControl GetSettingsUI()
      {
         if (_settingsUI == null)
         {
            _settingsUI = new LineChartStyleSettings();
            _settingsUI.numWidth.Value = _lineWidth;
         }
         return _settingsUI;
      }

      //Write settings value to host
      public void WriteSettings(ISettingsHost host)
      {
         host.Set("LineChartStyle.LineWidth", _lineWidth);
      }

      //---*** Private members and methods ***---

      //Private fields
      private LineChartStyleSettings _settingsUI;
      private int _lineWidth = 1;
   }
}
```

Fidelity Brokerage Services LLC, Member NYSE, SIPC
900 Salem Street, Smithfield, RI 02917