ComponentOne

Chart for WPF

Copyright © 2011 ComponentOne LLC. All rights reserved.

Corporate Headquarters
ComponentOne LLC
201 South Highland Avenue
3rd Floor
Pittsburgh, PA 15206 · USA

Internet: **info@ComponentOne.com**

Web site: **http://www.componentone.com**

Sales

E-mail: sales@componentone.com

Telephone: 1.800.858.2739 or 1.412.681.4343 (Pittsburgh, PA USA Office)

Trademarks

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

Warranty

ComponentOne warrants that the original CD (or diskettes) are free from defects in material and workmanship, assuming normal use, for a period of 90 days from the date of purchase. If a defect occurs during this time, you may return the defective CD (or disk) to ComponentOne, along with a dated proof of purchase, and ComponentOne will replace it at no charge. After 90 days, you can obtain a replacement for a defective CD (or disk) by sending it and a check for $25 (to cover postage and handling) to ComponentOne.

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

Copying and Distribution

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

This manual was produced using ComponentOne Doc-To-Help™.

Table of Contents

ComponentOne Chart for WPF Overview

ComponentOne Chart™ for WPF revolutionizes chart presentations through powerful rendering, rich styling elements, animations, and data-binding capabilities. Display your chart in a Smart Client application or take advantage of the XBAP support and extend your deployment capabilities to the Web.

With the built-in themes, chart types, and wide range of color palettes your customized chart is just clicks away – no additional coding required. Combine the simplicity of Chart for WPF with the power of WPF and you'll discover creating professionally designed chart types has never been easier.

Getting Started

If you're new to **Chart for WPF**, get started with the following topics:

- Chart for WPF Quick Start (page 31)
- Chart Types (page 66)
- Chart Data Series (page 94)

What's New in ComponentOne Chart for WPF

This documentation was last revised for 2011 v2 on June 14, 2011.

New Features

The following new features were added to **ComponentOne Chart for WPF** in the 2011 v2 release:

- **Added new enhancements to the ChartPanel Object**
  - Added protected method, ChartPanelObject.OnDataPointChanged
  - ChartPanelObject.AxisX and ChartPanelObject.AxisY properties were added to support auxliary axes.
  - Chart panel objects are hidden when its position is outside the plot area.

    Added ChartPanelObject.UseAxisLimits property. When UseAxisLimits = true, the object can not be moved outside the plot area.
- **Axis improvements**
  - Added GetAxisRect method that returns axis rectangle in chart control coordinates.
  - Added Axis.UseExactLimits that allows to turn off rounding for axis minimum and maximum.
- **Enabled caching for plot elements in .NET 4.0**

  Added CacheMode property that gets or sets the cache mode for plot elements. Only for .Net 4.

New Members

The following members were added to **Chart for WPF** in the 2011 v2 release:

Member	Description

OnDataPointChanged	Called when the data point was changed.
AxisX	Gets or sets the name of the X-Axis.
AxisY	Gets or sets the name of the Y-Axis.
UseAxisLimits	When UseAxisLimits = true, the object can not be moved outside the plot area.
GetAxisRect	Gets the coordinates of the axis rectangle.
UseExactLimits	Specifies whether to use exact data values for axis minimum and maximum. By default(UseExactLimits=false), the minimum and maximum can be adjusted to round numbers.
CacheMode	Gets or sets the cache mode for plot elements. Only for .Net 4.

New Topics

The following new topics were added to **Chart for WPF:**

New Topic	Description
Using MVVM (page 153)	Shows how to use C1Chart in a MVVM designed application.
XAML Quick Reference (page 42)	Shows how to use the WPF C1Chart using only XAML code.

New Task-Based Help Topics

The following task-based help topics were added to **Chart for WPF**:

Task-Based Help Topic	Description
Adding a Bar Series and a Line Series at the Same Time (page 168)	Shows how to programatically add a Bar series and a Line series at the same time..
Adding Connecting Lines to Prevent Pie Overlapping (page 172)	Shows how to add connecting lines using XAML code to prevent the Pie from overlapping.
Changing the Candle Stick Width (page 165)	Shows how to programatically change the candle stick width.
Changing the Offset for All Slices (page 172)	Shows how to programatically change the offset for all slices in the Pie chart.
Creating a HiLoOpenClose Chart (page 170)	Shows how to programmatically create a HiLoOpenClose chart.
Creating a Gaussian Curve (page 169)	Shows how to programatically create a Gaussian Curve.
Displaying Axis Labels on an Angle (page 159)	Shows how to programmatically display the axis labels on an angle.
Hiding the Chart Legend (page 162)	Shows how to use xaml to hide the chart legend.
Orienting the Data in the ChartLegend (page 163)	Shows how to orient the chart legend horizontally.
Scaling a Bubble Chart While Zooming (page 166)	Shows how to adjust the scale in the DataSeries.PlotElementLoaded event to scale the Bubble chart while zooming.
Scaling Both Independent Axes (page	Shows how to scale both independent axes using

167)	the C1Chart.PropertyChanged event.
Swapping X and Y Axes During Runtime (page 167)	Shows how to invert the X and Y axes after the chart was loaded.
Exporting Chart into a PDF Format (page 175)	Shows how to programatically export chart into a PDF format.
Setting the Default Viewing Angle for 3D Pie Chart (page 173)	Shows how to programatically set the default viewing angle for the 3D Pie chart.
Specifying the Color of Each Bar/Column in the Data Series (page 165)	Shows how to programatically specify the color of each bar/column in the datat series DataSeries.PlotElementLoaded event.
Zooming in C1Chart (page 167)	Shows how to add zooming behavior in C1Chart.

Tip: A version history containing a list of new features, improvements, fixes, and changes for each product is available on the ComponentOne Website at http://helpcentral.componentone.com/VersionHistory.aspx.

Revision History

The revision history provides recent enhancements to **Chart for WPF**.

What's New in 2011 v1

This documentation was last revised for 2011 v1 on February 22, 2011.

The following new features were added to **ComponentOne Chart for WPF** in the 2011 v1 release:

- **New Chart Types Added**

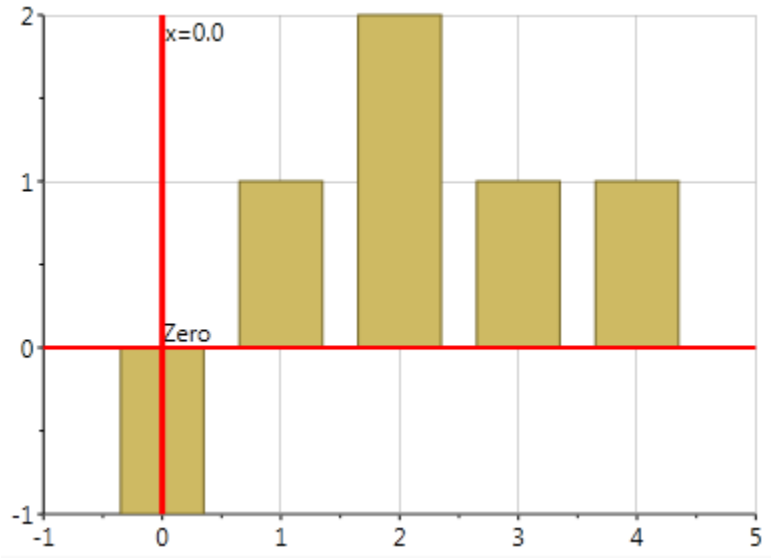
  Added **Polygon** and **PolygonFilled** chart types

  These are simialr to line/area, but they create a closed area defined by data points. These chart types (especially **PolygonFilled**) can be very handy when you need to add a zone with custom shape.

- **Add UI Elements Over the Chart**

  Added **ChartPanel** and **ChartPanelObject** classes that allows to position UI elements over the chart using data coordinates. For more information on these classes see Chart Panel (page 95).

  The following image illustrates the ChartPanelObjects:



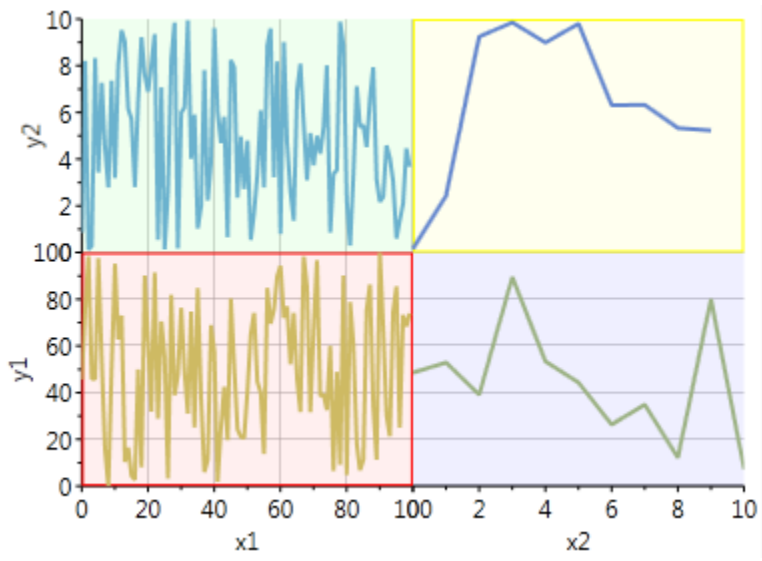
- **Add Symbols to Non-Symbol Charts**

  You can now apply **DataSeries.SymbolMarker** to non-symbol charts such as Area, Line, and Step.

- **Enhanced PlotArea Appearance**

  You can now add definitions in XAML using the **ChartView PlotAreaColumnDefinitions** and **PlotAreaRowDefinitions**. For more information see Plot Area Appearance (page 116).

  The following image illustrates the **ChartView PlotAreaColumnDefinitions** and **PlotAreaRowDefinitions** used to get the background colors for each plot area:



- **New Chart Samples**

  Added the following new samples for the new features: Advanced\Polygon, Interaction\AddRemove Markers, Interaction\Markders, Interaction\Mouse Marker, Combination\Plot Areas (vertically).

- **Multiple Plot Areas**

Added multiple plot areas (PlotArea class, ChartView.PlotAreas and Axis.PlotAreaIndex properties)

- **New Font Properties**

  Added font properties for the **Axis** class(FontFamily, FontSize, FontStyle, FontStretch, FontWeight).

- **Fast Render Mode**

  The rendering mode is controlled by the RenderMode property. The new rendering method shows a good performance with scatter charts.

- **Enhanced Symbol Charts**

  Added **LineAreaOptions.OptimizationRadiusScope** property and corresponding enum. The optimization can be applied to the symbol charts

  The **LineAreaOptions.OptimizationRadius** can be set on the data series

- **Axis Label Positioning**

  Improved axis label positioning for small time intervals

- **New Icons**

What's New in 2010 v3

This documentation was last revised for 2010 v3 on October 21, 2010.

Design-time support for **C1Chart3D** has been added. See the **ComponentOne 3D Chart for WPF** documentation for more information on this new control.

What's New in 2010 v2

New changes have been added to **ComponentOne Chart for WPF** in the 2010 v2 release:

New Changes

The following changes were added to **ComponentOne Chart for WPF** in the 2010 v2 release:

- The following properties of Axis class now are dependency properties: **AnnoPosition**, **LogBase**, **MajorUnit**, **MinorUnit**, **Position**, and **Scale**.

Installing Chart for WPF

The following sections provide helpful information on installing **Chart for WPF**.

Chart for WPF Setup Files

The installation program will create the directory **C:\Program Files\ComponentOne\Studio for WPF**, which contains the following subdirectories:

bin	Contains copies of all ComponentOne binaries (DLLs, EXEs). For **Chart for WPF**, the following .dlls are installed: C1.WPF.C1Chart.Design.dll C1.WPF.C1Chart.dll C1.WPF.C1Chart.Extended.dll C1.WPF.C1Chart3D.dll C1.WPF.C1Chart.Expression.Design.dll

	C1.WPF.C1Chart.VisualStudio.Design.dll
	C1.WPF.C1Chart.Design.4.0.dll
H2Help	Contains Microsoft Help 2.0 integrated documentation for all Studio components.
C1Chart\XAML	Contains the full XAML definitions of C1Chart styles and templates which can be used for creating your own custom styles and templates.

The ComponentOne Studio for WPF Help Setup program installs integrated Microsoft Help 2.0 and Microsoft Help Viewer help to the C:\Program Files\ComponentOne\Studio for WPF directory in the following folders:

H2Help	Contains Microsoft Help 2.0 integrated documentation for all Studio components.
HelpViewer	Contains Microsoft Help Viewer Visual Studio 2010 integrated documentation for all Studio components.

Samples

Samples for the product are installed in the **ComponentOne Samples** folder by default. The path of the **ComponentOne Samples** directory is slightly different on Windows XP and Windows 7/Vista machines:

Windows XP path: C:\Documents and Settings\<username>\My Documents\ComponentOne Samples

Windows 7/Vista path: C:\Users\<username>\Documents\ComponentOne Samples

The **ComponentOne Samples** folder contains the following subdirectories:

Common	Contains support and data files that are used by many of the demo programs.
C1Chart	Contains samples for **Chart for WPF**.

Samples can be accessed from the **ComponentOne Studio for WPF ControlExplorer**. On your desktop, click the **Start** button and then click **All Programs| ComponentOne | Studio for WPF | C1WPFChart Samples**.

System Requirements

System requirements include the following:

Operating Systems:	Microsoft Windows® XP with Service Pack 2 (SP2)
	Windows Vista™
	Windows Server® 2003
	Windows Server 2008
	Windows 7
Environments:	.NET Framework 3.5 or later
	Visual Studio® 2005 extensions for .NET Framework 2.0 November 2006 CTP

Microsoft® Expression® Blend Compatibility:	**Chart for WPF** has special design-time support for Blend, which includes workarounds for correct XAML serialization and a customizable user interface.

Note: The **C1.WPF.VisualStudio.Design.dll** assembly is required by Visual Studio 2008 and the **C1.WPF.Expression.Design.dll** assembly is required by Expression Blend. The **C1.WPF.Expression.Design.dll** and **C1.WPF.VisualStudio.Design.dll** assemblies installed with **Chart for WPF** should always be placed in the same folder as **C1.WPF.C1Chart.dll**; the DLLs should not be placed in the Global Assembly Cache (GAC).

Installing Demonstration Versions

If you wish to try **ComponentOne Chart for WPF** and do not have a serial number, follow the steps through the installation wizard and use the default serial number.

The only difference between unregistered (demonstration) and registered (purchased) versions of our products is that registered versions will stamp every application you compile so a ComponentOne banner will not appear when your users run the applications.

Uninstalling Chart for WPF

To uninstall **ComponentOne Chart for WPF**:

1. Open the **Control Panel** and select **Add or Remove Programs** (**Programs and Features** in Vista).
2. Select **ComponentOne Studio for WPF** and click the **Remove** button.
3. Click **Yes** to remove the program.

End-User License Agreement

All of the ComponentOne licensing information, including the ComponentOne end-user license agreements, the ComponentOne licensing model, and frequently asked licensing questions, is available online at http://www.componentone.com/SuperPages/Licensing/.

Licensing FAQs

This section describes the main technical aspects of licensing. It may help the user to understand and resolve licensing problems he may experience when using ComponentOne .NET and ASP.NET products.

What is Licensing?

Licensing is a mechanism used to protect intellectual property by ensuring that users are authorized to use software products.

Licensing is not only used to prevent illegal distribution of software products. Many software vendors, including ComponentOne, use licensing to allow potential users to test products before they decide to purchase them.

Without licensing, this type of distribution would not be practical for the vendor or convenient for the user. Vendors would either have to distribute evaluation software with limited functionality, or shift the burden of managing software licenses to customers, who could easily forget that the software being used is an evaluation version and has not been purchased.

How does Licensing Work?

ComponentOne uses a licensing model based on the standard set by Microsoft, which works with all types of components.

Note: The **Compact Framework** components use a slightly different mechanism for run-time licensing than the other ComponentOne components due to platform differences.

When a user decides to purchase a product, he receives an installation program and a Serial Number. During the installation process, the user is prompted for the serial number that is saved on the system. (Users can also enter the serial number by clicking the **License** button on the **About Box** of any ComponentOne product, if available, or by rerunning the installation and entering the serial number in the licensing dialog.)

When a licensed component is added to a form or Web page, Visual Studio obtains version and licensing information from the newly created component. When queried by Visual Studio, the component looks for licensing information stored in the system and generates a run-time license and version information, which Visual Studio saves in the following two files:

- An assembly resource file which contains the actual run-time license
- A "licenses.licx" file that contains the licensed component strong name and version information

These files are automatically added to the project.

In WinForms and ASP.NET 1.x applications, the run-time license is stored as an embedded resource in the assembly hosting the component or control by Visual Studio. In ASP.NET 2.x applications, the run-time license may also be stored as an embedded resource in the App_Licenses.dll assembly, which is used to store all run-time licenses for all components directly hosted by WebForms in the application. Thus, the App_licenses.dll must always be deployed with the application.

The licenses.licx file is a simple text file that contains strong names and version information for each of the licensed components used in the application. Whenever Visual Studio is called upon to rebuild the application resources, this file is read and used as a list of components to query for run-time licenses to be embedded in the appropriate assembly resource. Note that editing or adding an appropriate line to this file can force Visual Studio to add run-time licenses of other controls as well.

Note that the licenses.licx file is usually not shown in the Solution Explorer; it appears if you press the **Show All Files** button in the Solution Explorer's toolbox, or from Visual Studio's main menu, select **Show All Files** on the **Project** menu.

Later, when the component is created at run time, it obtains the run-time license from the appropriate assembly resource that was created at design time and can decide whether to simply accept the run-time license, to throw an exception and fail altogether, or to display some information reminding the user that the software has not been licensed.

All ComponentOne products are designed to display licensing information if the product is not licensed. None will throw licensing exceptions and prevent applications from running.

Common Scenarios

The following topics describe some of the licensing scenarios you may encounter.

Creating components at design time

This is the most common scenario and also the simplest: the user adds one or more controls to the form, the licensing information is stored in the licenses.licx file, and the component works.

Note that the mechanism is exactly the same for Windows Forms and Web Forms (ASP.NET) projects.

Creating components at run time

This is also a fairly common scenario. You do not need an instance of the component on the form, but would like to create one or more instances at run time.

In this case, the project will not contain a licenses.licx file (or the file will not contain an appropriate run-time license for the component) and therefore licensing will fail.

To fix this problem, add an instance of the component to a form in the project. This will create the licenses.licx file and things will then work as expected. (The component can be removed from the form after the licenses.licx file has been created).

Adding an instance of the component to a form, then removing that component, is just a simple way of adding a line with the component strong name to the licenses.licx file. If desired, you can do this manually using notepad or Visual Studio itself by opening the file and adding the text. When Visual Studio recreates the application resources, the component will be queried and its run-time license added to the appropriate assembly resource.

Inheriting from licensed components

If a component that inherits from a licensed component is created, the licensing information to be stored in the form is still needed. This can be done in two ways:

- Add a LicenseProvider attribute to the component.

  This will mark the derived component class as licensed. When the component is added to a form, Visual Studio will create and manage the licenses.licx file, and the base class will handle the licensing process as usual. No additional work is needed. For example:

```
[LicenseProvider(typeof(LicenseProvider))]
class MyGrid: C1.Win.C1FlexGrid.C1FlexGrid
{
  // ...
}
```

- Add an instance of the base component to the form.

  This will embed the licensing information into the licenses.licx file as in the previous scenario, and the base component will find it and use it. As before, the extra instance can be deleted after the licenses.licx file has been created.

Please note, that C1 licensing will not accept a run time license for a derived control if the run time license is embedded in the same assembly as the derived class definition, and the assembly is a DLL. This restriction is necessary to prevent a derived control class assembly from being used in other applications without a design time license. If you create such an assembly, you will need to take one of the actions previously described create a component at run time.

Using licensed components in console applications

When building console applications, there are no forms to add components to, and therefore Visual Studio won't create a licenses.licx file.

In these cases, create a temporary Windows Forms application and add all the desired licensed components to a form. Then close the Windows Forms application and copy the licenses.licx file into the console application project.

Make sure the licenses.licx file is configured as an embedded resource. To do this, right-click the licenses.licx file in the Solution Explorer window and select **Properties**. In the property window, set the **Build Action** property to **Embedded Resource**.

Using licensed components in Visual C++ applications

There is an issue in VC++ 2003 where the licenses.licx is ignored during the build process; therefore, the licensing information is not included in VC++ applications.

To fix this problem, extra steps must be taken to compile the licensing resources and link them to the project. Note the following:

1. Build the C++ project as usual. This should create an exe file and also a licenses.licx file with licensing information in it.
2. Copy the licenses.licx file from the app directory to the target folder (Debug or Release).

3. Copy the C1Lc.exe utility and the licensed dlls to the target folder. (Don't use the standard lc.exe, it has bugs.)
4. Use C1Lc.exe to compile the licenses.licx file. The command line should look like this:
```
c1lc /target:MyApp.exe /complist:licenses.licx /i:C1.Win.C1FlexGrid.dll
```
5. Link the licenses into the project. To do this, go back to Visual Studio, right-click the project, select properties, and go to the Linker/Command Line option. Enter the following:
```
/ASSEMBLYRESOURCE:Debug\MyApp.exe.licenses
```
6. Rebuild the executable to include the licensing information in the application.

Using licensed components with automated testing products

Automated testing products that load assemblies dynamically may cause them to display license dialogs. This is the expected behavior since the test application typically does not contain the necessary licensing information, and there is no easy way to add it.

This can be avoided by adding the string "C1CheckForDesignLicenseAtRuntime" to the AssemblyConfiguration attribute of the assembly that contains or derives from ComponentOne controls. This attribute value directs the ComponentOne controls to use design time licenses at run time.

For example:
```
    #if AUTOMATED_TESTING
[AssemblyConfiguration("C1CheckForDesignLicenseAtRuntime")]
    #endif
public class MyDerivedControl : C1LicensedControl
{
      // ...
}
```

Note that the AssemblyConfiguration string may contain additional text before or after the given string, so the AssemblyConfiguration attribute can be used for other purposes as well. For example:
```
[AssemblyConfiguration("C1CheckForDesignLicenseAtRuntime,BetaVersion")]
```

THIS METHOD SHOULD ONLY BE USED UNDER THE SCENARIO DESCRIBED. It requires a design time license to be installed on the testing machine. Distributing or installing the license on other computers is a violation of the EULA.

Troubleshooting

We try very hard to make the licensing mechanism as unobtrusive as possible, but problems may occur for a number of reasons.

Below is a description of the most common problems and their solutions.

I have a licensed version of a ComponentOne product but I still get the splash screen when I run my project.

If this happens, there may be a problem with the licenses.licx file in the project. It either doesn't exist, contains wrong information, or is not configured correctly.

First, try a full rebuild (**Rebuild All** from the Visual Studio **Build** menu). This will usually rebuild the correct licensing resources.

If that fails follow these steps:

1. Open the project and go to the Solution Explorer window.
2. Click the **Show All Files** button on the top of the window.
3. Find the licenses.licx file and open it. If prompted, continue to open the file.

4. Change the version number of each component to the appropriate value. If the component does not appear in the file, obtain the appropriate data from another licenses.licx file or follow the alternate procedure following.
5. Save the file, then close the licenses.licx tab.
6. Rebuild the project using the **Rebuild All** option (not just **Rebuild**).

Alternatively, follow these steps:

1. Open the project and go to the Solution Explorer window.
2. Click the **Show All Files** button on the top of the window.
3. Find the licenses.licx file and delete it.
4. Close the project and reopen it.
5. Open the main form and add an instance of each licensed control.
6. Check the Solution Explorer window, there should be a licenses.licx file there.
7. Rebuild the project using the **Rebuild All** option (not just **Rebuild**).

For ASP.NET 2.x applications, follow these steps:

1. Open the project and go to the Solution Explorer window.
2. Find the licenses.licx file and right-click it.
3. Select the Rebuild Licenses option (this will rebuild the App_Licenses.licx file).
4. Rebuild the project using the **Rebuild All** option (not just **Rebuild**).

I have a licensed version of a ComponentOne product on my Web server but the components still behave as unlicensed.

There is no need to install any licenses on machines used as servers and not used for development.

The components must be licensed on the development machine, therefore the licensing information will be saved into the executable (exe or dll) when the project is built. After that, the application can be deployed on any machine, including Web servers.

For ASP.NET 2.x applications, be sure that the App_Licenses.dll assembly created during development of the application is deployed to the bin application bin directory on the Web server.

If your ASP.NET application uses WinForms user controls with constituent licensed controls, the runtime license is embedded in the WinForms user control assembly. In this case, you must be sure to rebuild and update the user control whenever the licensed embedded controls are updated.

I downloaded a new build of a component that I have purchased, and now I'm getting the splash screen when I build my projects.

Make sure that the serial number is still valid. If you licensed the component over a year ago, your subscription may have expired. In this case, you have two options:

Option 1 - Renew your subscription to get a new serial number.

If you choose this option, you will receive a new serial number that you can use to license the new components (from the installation utility or directly from the **About Box**).

The new subscription will entitle you to a full year of upgrades and to download the latest maintenance builds directly from http://prerelease.componentone.com/.

Option 2 – Continue to use the components you have.

Subscriptions expire, products do not. You can continue to use the components you received or downloaded while your subscription was valid.

Technical Support

ComponentOne offers various support options. For a complete list and a description of each, visit the ComponentOne Web site at http://www.componentone.com/SuperProducts/SupportServices/.

Some methods for obtaining technical support include:

- Online Resources

  ComponentOne provides customers with a comprehensive set of technical resources in the form of FAQs, samples and videos, Version Release History, searchable Knowledge base, searchable Online Help and more. We recommend this as the first place to look for answers to your technical questions.

- **Online Support via our Incident Submission Form**

  This online support service provides you with direct access to our Technical Support staff via an online incident submission form. When you submit an incident, you'll immediately receive a response via e-mail confirming that you've successfully created an incident. This e-mail will provide you with an Issue Reference ID and will provide you with a set of possible answers to your question from our Knowledgebase. You will receive a response from one of the ComponentOne staff members via e-mail in 2 business days or less.

- **Product Forums**

  ComponentOne's **product forums** are available for users to share information, tips, and techniques regarding ComponentOne products. ComponentOne developers will be available on the forums to share insider tips and technique and answer users' questions. Please note that a ComponentOne User Account is required to participate in the ComponentOne Product Forums.

- **Installation Issues**

  Registered users can obtain help with problems installing ComponentOne products. Contact technical support by using the online incident submission form or by phone (412.681.4738). Please note that this does not include issues related to distributing a product to end-users in an application.

- **Documentation**

  Microsoft integrated ComponentOne documentation can be installed with each of our products, and documentation is also available online. If you have suggestions on how we can improve our documentation, please email the Documentation team. Please note that e-mail sent to the Documentation team is for documentation feedback only. Technical Support and Sales issues should be sent directly to their respective departments.

Note: You must create a ComponentOne Account and register your product with a valid serial number to obtain support using some of the above methods.

Redistributable Files

ComponentOne Chart for WPF is developed and published by ComponentOne LLC. You may use it to develop applications in conjunction with Microsoft Visual Studio or any other programming environment that enables the user to use and integrate the control(s). You may also distribute, free of royalties, the following Redistributable Files with any such application you develop to the extent that they are used separately on a single CPU on the client/workstation side of the network:

- C1.WPF.C1Chart.dll
- C1.WPF.C1Chart.Extended.dll

Note: Both the C1.WPF.C1Chart.dll and C1.WPF.C1Chart.Expression.Design.dll files are installed to **C:\Program Files\ComponentOne\Studio for WPF\bin** by default. The **C1.WPF.C1Chart.Expression.Design.dll** installed with **Chart for WPF for WPF** should always be placed in the same folder as the **C1.WPF.C1Chart.dll**; these files should NOT be placed in the GAC. The **C1.WPF.C1Chart.Expression.Design.dll** assembly is required by Blend.

Site licenses are available for groups of multiple developers. Please contact Sales@ComponentOne.com for details.

About This Documentation

You can create your chart applications using Microsoft Expression Blend or Visual Studio 2008, but Blend is currently the only design-time environment that allows users to design XAML documents visually.

Acknowledgements

Microsoft, Windows, Windows Vista, Visual Studio, and Microsoft Expression are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

ComponentOne

If you have any suggestions or ideas for new features or controls, please call us or write:

Corporate Headquarters

ComponentOne LLC
201 South Highland Avenue
3rd Floor
Pittsburgh, PA 15206 • USA
412.681.4343
412.681.4384 (Fax)

http://www.componentone.com

ComponentOne Doc-To-Help

This documentation was produced using ComponentOne Doc-To-Help® Enterprise.

XAML and XAML Namespaces

XAML is a declarative XML-based language that is used as a user interface markup language in Windows Presentation Foundation and the .NET Framework 3.0. With XAML you can create a graphically rich customized user interface, perform databinding, and much more. For more information on XAML and the .NET Framework 3.0, please see http://www.microsoft.com.

XAML Namespaces

Namespaces organize the objects defined in an assembly. Assemblies can contain multiple namespaces, which can in turn contain other namespaces. Namespaces prevent ambiguity and simplify references when using large groups of objects such as class libraries.

When you create a Microsoft Blend project, a XAML file is created for you, and some initial namespaces are specified:

Namespace	Description
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"	This is the default Windows Presentation Foundation namespace.
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"	This is a XAML namespace that is mapped to the **x:** prefix. The **x:** prefix provides a quick, easy way to reference the namespace, which defines many commonly-used features necessary for WPF applications.

When you add a C1Chart control to the window in Microsoft Expression Blend, **Blend** automatically creates a clr-namespace, or common language runtime (CLR) namespace that maps to the C1.WPF.C1Chart assembly. The namespace looks like the following:

```
xmlns:c1chart="http://schemas.componentone.com/xaml/c1chart"
```

The clr-namespace value is **C1.WPF.C1Chart**, and the assembly value is **C1.WPF.C1Chart**.

When you add a **C1Chart** control to the window in Visual Studio 2008, Visual Studio automatically creates a clr-namespace, or common run time (CLR) namespace that maps to the **C1.WPF.C1Chart** assembly. The namespace looks like the following:

```
xmlns:c1chart="http://schemas.componentone.com/xaml/c1chart"
```

You can also choose to create your own custom name for the namespace. For example:

```
xmlns:MyC1Chart="clr-namespace:C1.WPF.C1Chart;assembly=C1.WPF.C1Chart"
```

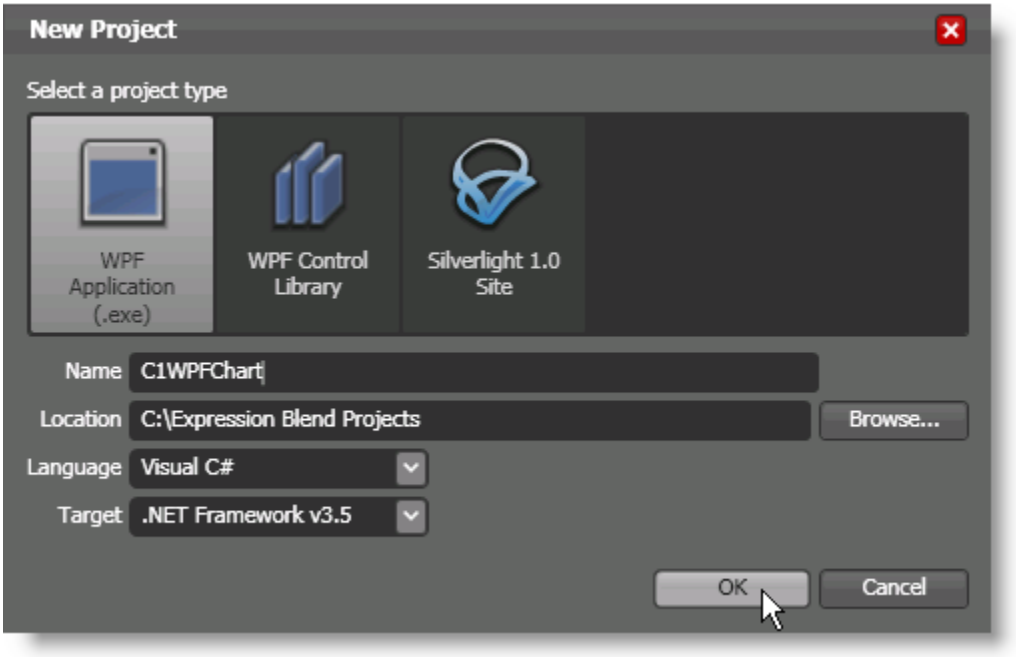
You can now use your custom namespace when assigning properties, methods and events.

Creating a Microsoft Blend Project

To create a new Blend project, complete the following steps:

1. From the File menu, select **New Project** or click **New Project** in the Blend startup window. The **Create New Project** dialog box opens.
2. Select the **WPF Application (.exe)**, and enter a name for the project in the **Name** text box.

   The **WPF Application (.exe)** creates a project for a Windows-based application that can be built and run while being designed.
3. Select the **Browse** button to specify a location for the project.
4. Select a language from the **Language** drop-down box and click **OK**.



A new Blend project with a XAML window is created.

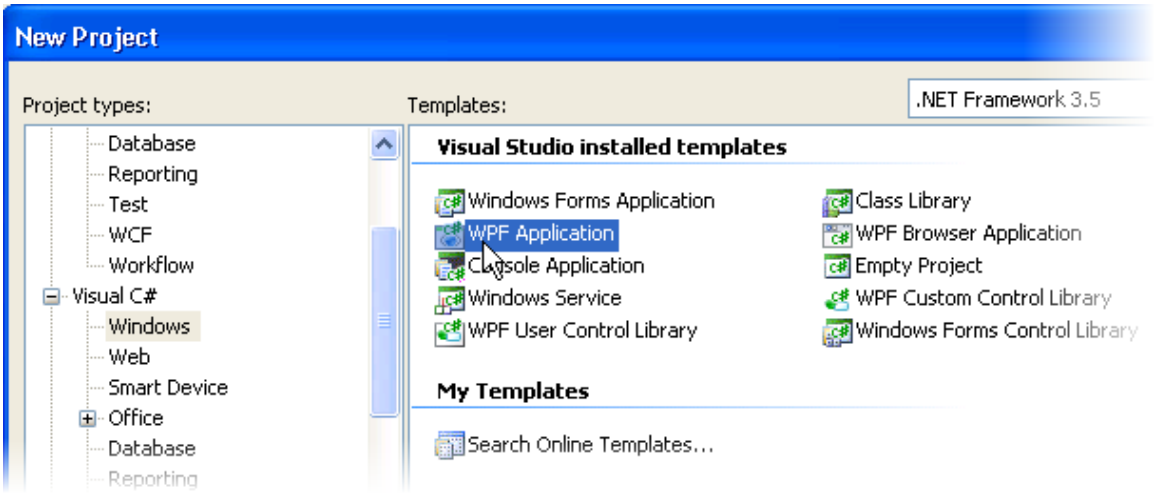
Creating a .NET Project in Visual Studio

To create a new .NET project, open Visual Studio 2008 and complete the following steps:

1. From the **File** menu in Microsoft Visual Studio 2008, select **New Project**. The **New Project** dialog box opens.
2. Choose the appropriate .NET Framework from the drop-down box in the top-right of the dialog box.
3. Under **Project Types**, select either **Visual Basic** or **Visual C#**.

> **Note:** In Visual Studio 2005 select **NET Framework 3.0** under **Visual Basic** or **Visual C#** in the Project types tree.

4. Select **WPF Application** from the list of **Templates** in the right pane.



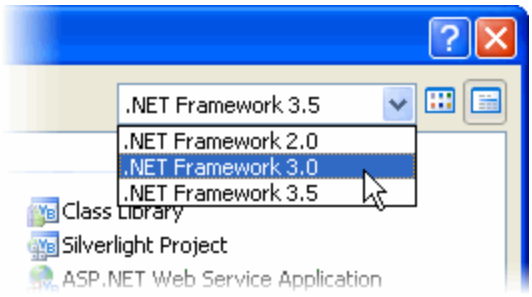
5. Enter a name for your application in the **Name** field and click **OK**. A new Microsoft Visual Studio .NET WPF project is created with a .xaml file that will be used to define your user interface and commands in the application.

> **Note:** You can create your chart applications using Microsoft Expression Blend or Visual Studio, but Blend is currently the only design-time environment that allows users to design XAML documents visually.

Creating an XAML Browser Application (XBAP) in Visual Studio

To create a new XAML Browser Application (XBAP) in Visual Studio 2008, complete the following steps:

1. From the **File** menu in Microsoft Visual Studio 2008, select **New Project**. The **New Project** dialog box opens.
2. Choose the appropriate .NET Framework from the Framework drop-down box in the top-right of the dialog box.



3. Under Project types, select either **Visual Basic** or **Visual C#**.
4. Choose **WPF Browser Application** from the list of **Templates** in the right pane.

   Note: If using Visual Studio 2005, you may need to select **XAML Browser Application (WPF)** after selecting **NET Framework 3.0** under **Visual Basic** or **Visual C#** in the left-side menu.

5. Enter a name for your application in the **Name** field and click **OK**.

   A new Microsoft Visual Studio .NET WPF Browser Application project is created with a XAML file that will be used to define your user interface and commands in the application.

When you run this application to a Web server, users can locate the URL and run your program from within the IE Sandbox.

Note: The XAML Browser Application only works in IE 6 or IE 7.

Adding the Chart for WPF Components to a Blend Project

In order to use C1Chart or another **Chart for WPF** component in the Design workspace of Blend, you must first add a reference to the **C1.WPF.C1Chart** assembly and then add the component from Blend's **Asset Library**.

To add a reference to the assembly:

1. Click on **Project | Add Reference**.
2. Browse to find the **C1.WPF.C1Chart.dll** installed with Chart for WPF.

   Note: The C1.WPF.C1Chart.dll is installed to **C:\Program Files\ComponentOne\Studio for WPF\bin** by default.

3. Select **C1.WPF.C1Chart.dll** and click **Open**. A reference is added to your project.

To add a component from the Asset Library:

1. Once you have added a reference to the **C1.WPF.C1Chart** assembly, click the **Asset Library** button in the Blend toolbox. The **Asset Library** appears.
2. Click the **Custom Controls** tab. All of the **Chart for WPF** main and auxiliary components are listed here.
3. Select **C1Chart** from the **Asset Library** dialog box. The component appears in the Toolbox above the **Asset Library** button.
4. Double-click the **C1Chart** component in the Toolbox to add it to the Window1.xaml.

Adding the Chart for WPF Components to a Visual Studio Project

When you install **Chart for WPF** the C1Chart control should be added to your Visual Studio Toolbox. You can also manually add ComponentOne controls to the Toolbox.

ComponentOne Chart for WPF provides the following controls:

- C1Chart
- C1ChartLegend

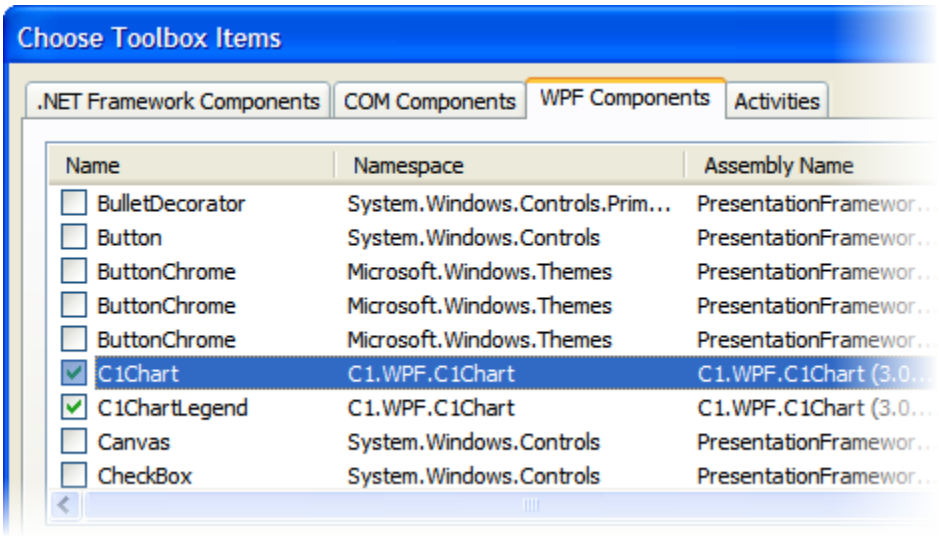
To use **Chart for WPF**, add the CChart control to the window or add a reference to the **C1.WPF.C1Chart** assembly to your project.

Manually Adding Chart for WPF to the Toolbox

To manually add the **C1Chart** control to the Visual Studio Toolbox, complete the following steps:

1. Open the Visual Studio IDE (Microsoft Development Environment). Make sure the Toolbox is visible (select **Toolbox** in the **View** menu, if necessary) and right-click the Toolbox to open its context menu.
2. To make **Chart for WPF, C1Chart** component appear on its own tab in the Toolbox, select **Add Tab** from the context menu and type in the tab name, **Chart for WPF**, for example.
3. Right-click the tab where the component is to appear and select **Choose Items** from the context menu.

   The **Choose Toolbox Items** dialog box opens.
4. In the dialog box, select the **WPF Components** tab. Sort the list by Namespace (click the *Namespace* column header) and select the check boxes for all components belonging to the C1.WPF.C1Chart namespace. Note that there may be more than one component for each namespace.



Adding Chart for WPF to the Window

To add **Chart for WPF** to a window or page, complete the following steps:

1. Add the **C1Chart** control to the Visual Studio Toolbox.
2. Double-click **C1Chart** or drag the control onto the window.

Adding a Reference to the Assembly

To add a reference to the **Chart for WPF** assembly, complete the following steps:

1. Select the **Add Reference** option from the **Project** menu of your project.
2. Select the **ComponentOne Chart for WPF** assembly from the list on the **.NET** tab or on the **Browse** tab, browse to find the C1.WPF.C1Chart.dll file and click **OK**.
3. Double-click the window caption area to open the code window. At the top of the file, add the following **Imports** statements (**using** in C#):

```
Imports C1.WPF.C1Chart
```

   This makes the objects defined in the **Chart for WPF** assembly visible to the project.

Key Features

Chart for WPF provides the following unique features:

- **Universal and Flexible Data Binding**

  Easily bind the **Chart for WPF** control to a data table, collection of business objects, or XML file by setting a few properties.

- **Extend Your Deployment Capabilities with XBAP Support**

  Chart for WPF is fully compatible with the new XBAP deployment capabilities of Windows Presentation Foundation. The XBAP deployment allows publishing to the client's (supported) browser for a full featured application without a Windows installation.

- **Choose From Over 35 Built-in Chart Types**

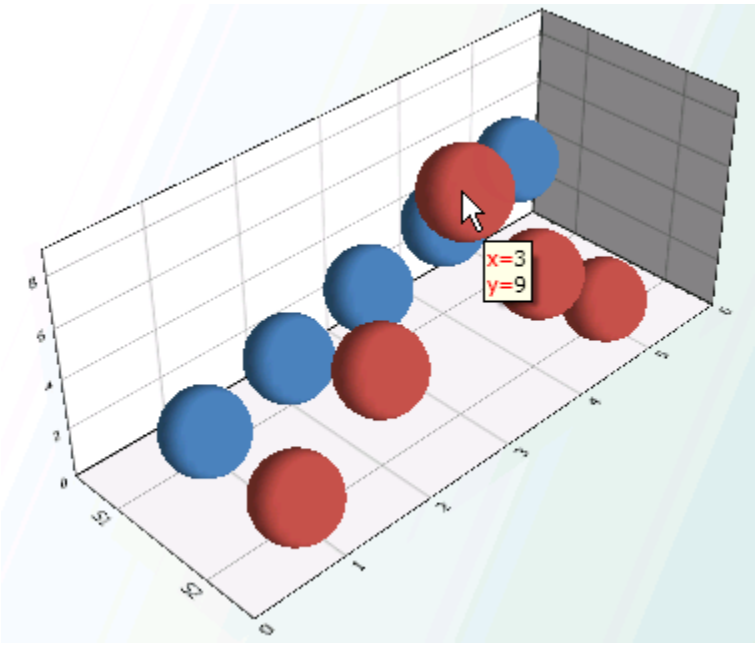
  Choose from the standard chart types (bar charts, line charts, pie charts, area charts, and more) or select from more advanced chart types such as a 3D doughnut pie or 3D ribbon.

- **Improve the Readability of Your Chart with Data Labels**

  Highlight important data points and provide valuable data information by adding data labels on each data series. There is no limit to the number of data labels a chart can contain.

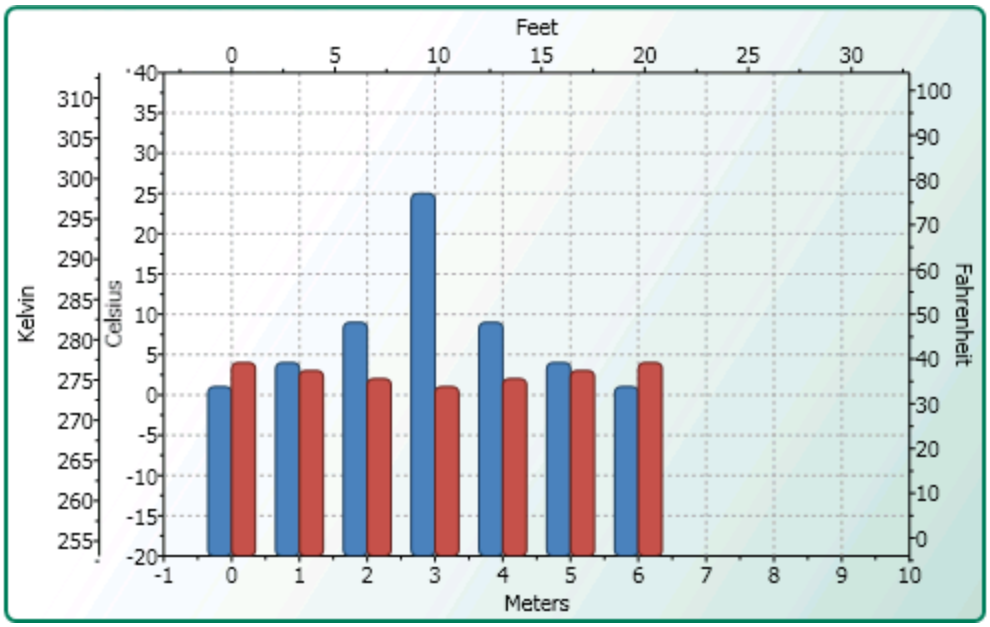
- **Charts That Come Alive with ToolTips**

  Display specific text when the user mouses over chart elements. **Chart for WPF** gives you the control to format the information displayed in the ToolTip.



- **Display Highly Representative Data with Multiple Axes**

  Place multiple axes to the top, bottom, left, or right of the chart to create a more detailed representation of data.

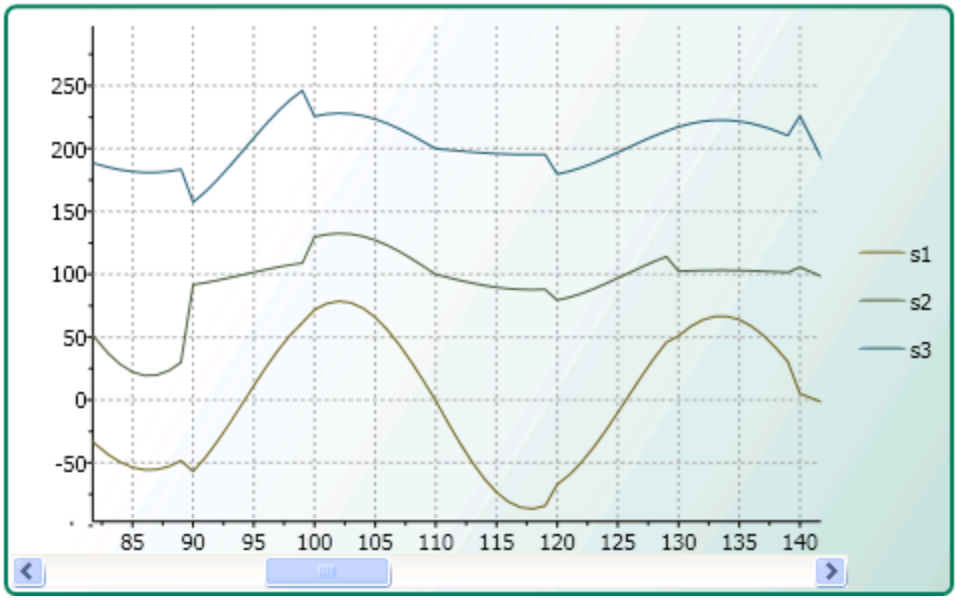


- **Highly Interactive End-user Experience**

  Chart for WPF provides a wide range of interactive behaviors ranging from built-in tools such as scrolling, scaling, and rotating charts at run time.

- **Display Large Amounts of Data with Axis Scrolling**

  Use the Scale and Value properties to enable scrolling along the chart axes. This is helpful when you have large amounts of X or Y data to display.



- **Create Dramatic Data Presentation with Multiple Charts**

  Chart for WPF allows you to combine various chart types, on a single plot, to create dramatic data presentation.

- **Highly Customizable Plot Elements**

  Quickly and efficiently change the appearance of any plot element using the symbol and connection properties.

- **Create a Wealth of Custom Palettes for Plot Elements**

  Experience more customizing options for plot elements; you are not restricted to a predefined palette. For example, use the CustomPalette property to define various brushes for plot elements.

- **Industry Leading Stacking Charts**

  Line, area, bar, and column charts can be stacked to display more complex data in a smaller space.

- **12 built-in themes available, giving you the means to achieve colorful, professional-looking applications**

  Customize the appearance of your chart application with built-in Office 2007, Vista, and Office 2003 themes, or create your own themes based on the included themes. Your polished chart is just a click away.

- **Enhanced graphic effects**

  Add some flair to your chart by enabling bitmap effects such as bevel, shadow, blur, glow, and emboss.

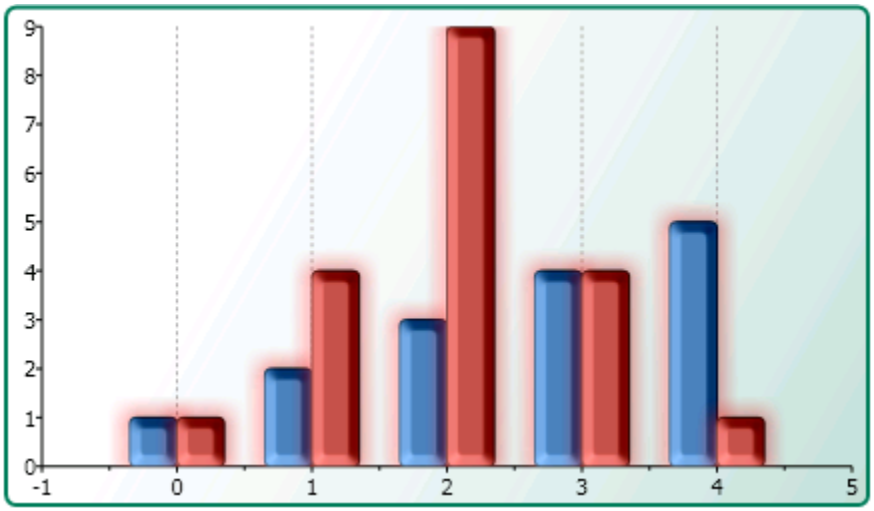


Chart for WPF Quick Start

The following quick start guide is intended to get you up and running with **Chart for WPF**. This quick start will guide you through the four basic steps for creating a typical chart such as a Bar chart: choose the chart, add one or more data series to the chart, set up and format the axes, and adjust the chart's appearance using the Theme property.

Step 1 of 4: Adding Chart for WPF to your Project

In this step you'll either begin in Visual Studio or Blend to create a chart application using **Chart for WPF**. When you add the **C1Chart** control to your Visual Studio or Blend project you'll have a functional column chart with fake data. If you use XAML code only to initialize your chart control you'll have an empty chart without the fake data.

To add Chart for WPF to your Visual Studio Project using XAML:

1. Create a new WPF project in Visual Studio.
2. Add a reference to the **C1.WPF.C1Chart** assembly. In the Solution Explorer right-click on **References** and select **Add Reference**. In the **Add Reference** dialog box select the Browse tab. Browse for the **C1.WPF.C1Chart.dll** and select **OK**.
3. Define the **System** and the **C1.WPF.C1Chart** prefixes.

```
xmlns:System="clr-namespace:System;assembly=mscorlib"
        xmlns:c1chart="clr-
namespace:C1.WPF.C1Chart;assembly=C1.WPF.C1Chart"
```

4. Inside the Grid, initialize the C1Chart control.

```
<c1chart:C1Chart Margin="0,0,8,8" MinHeight="160" MinWidth="240"
Content="C1Chart">
</c1chart:C1Chart>
```

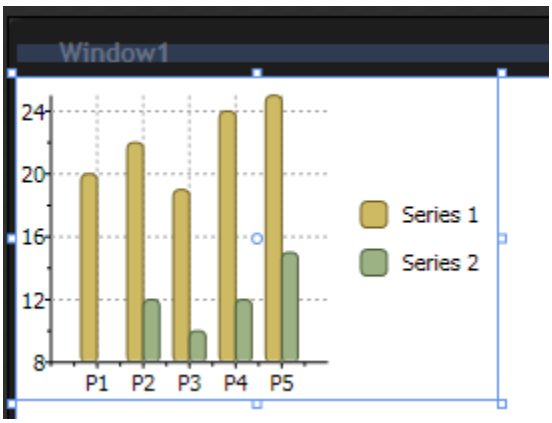
The chart appears empty since we did not add the data for it yet.

In the next step, Step 2 of 4: Adding Data to the Chart (page 32), you will add the data for C1Chart.

To add Chart for WPF to your Blend Project:

1. Create a new WPF project in Blend.
2. Add the **C1Chart** control to your window.

The **C1Chart** control is added to the Window.



The default "fake" data is added to the Chart control.

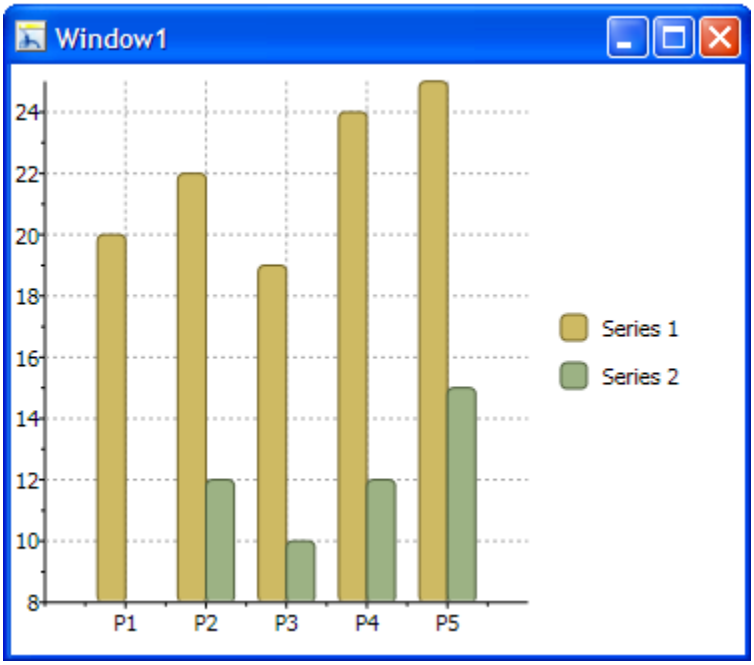
```
<my:C1Chart Margin="10,10,68,102" Name="c1Chart1">
            <my:C1Chart.Data>
                <my:ChartData>
                    <my:ChartData.ItemNames>P1 P2 P3 P4
P5</my:ChartData.ItemNames>
                    <my:DataSeries Label="Series 1" Values="20 22 19 24
25" />
                    <my:DataSeries Label="Series 2" Values="8 12 10 12 15"
/>
                </my:ChartData>
            </my:C1Chart.Data>
            <my:C1ChartLegend DockPanel.Dock="Right" />
        </my:C1Chart>
```

3. Resize the **C1Chart** control so it fills up the Window.

   In the next step, Step 2 of 4: Adding Data to the Chart (page 32), you will add the data for C1Chart.

Run the program and observe:

The **C1Chart** control will appear like the following if you added the chart control to your Blend or Visual Studio project. If you just added the Chart control using XAML code then the Chart will appear empty since default data was not added to it.



You've successfully created a chart application. In the next step you'll customize the data series for the **C1Chart** control.

Step 2 of 4: Adding Data to the Chart

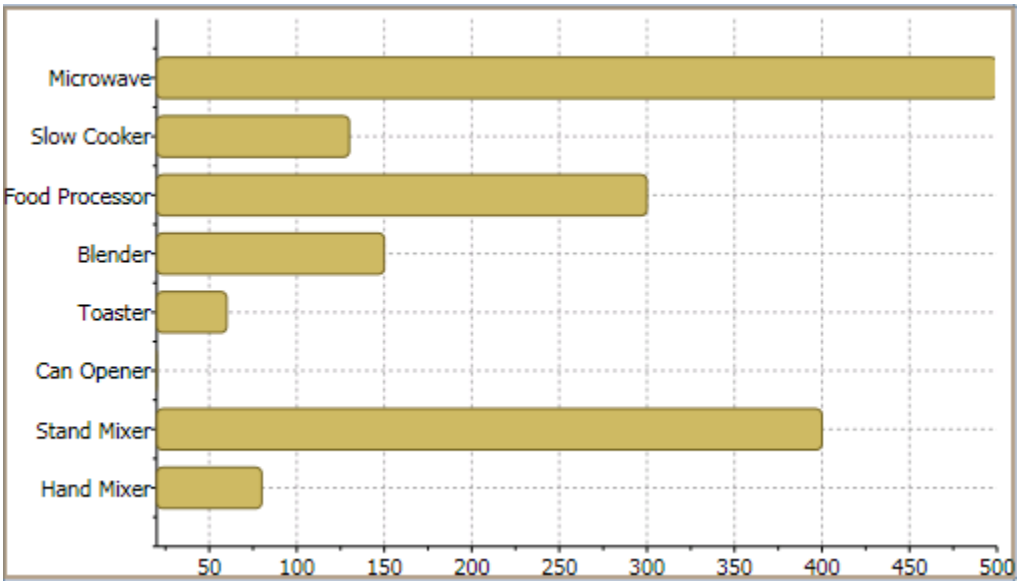
In the last step, you added the **C1Chart** control to the Window. In this step, you will add a **DataSeries** object and data for it.

To add data to the chart in Visual Studio using XAML

1. Set the ChartType property to **Bar** within the "<c1chart:C1Chart></c1chart:C1Chart>" tag using the following XAML code:
```
ChartType="Bar"
```
2. Add the data for the chart control using the following XAML code:
```
<c1chart:C1Chart.Data>
                <c1chart:ChartData>
                    <c1chart:ChartData.ItemNames>
                        <x:Array Type="{x:Type System:String}">
                            <System:String>Hand Mixer</System:String>
                            <System:String>Stand Mixer</System:String>
                            <System:String>Can Opener</System:String>
                            <System:String>Toaster</System:String>
                            <System:String>Blender</System:String>
                            <System:String>Food Processor</System:String>
                            <System:String>Slow Cooker</System:String>
                            <System:String>Microwave</System:String>
                        </x:Array>
                    </c1chart:ChartData.ItemNames>
                    <c1chart:DataSeries Values="80 400 20 60 150 300 130
500" AxisX="Price" AxisY="Kitchen Electronics" Label="Price"/>
                </c1chart:ChartData>
            </c1chart:C1Chart.Data>
```

In this step we use one **DataSeries** that has eight X-values. We added **ItemNames** of the type string to the **ChartData** to represent the string name for each data value. We used an array of string names for the **ItemNames** since a few of the item names included spaces. We were able to use the System:String namespace since we declared the namespace for it in step 4 of Step 1 of 4: Adding Chart for WPF to your Project (page 31).

The new data appears on the chart like the following:



In the next step, Step 3 of 4: Format the Axes (page 38), you'll learn how to customize the axes using XAML code.

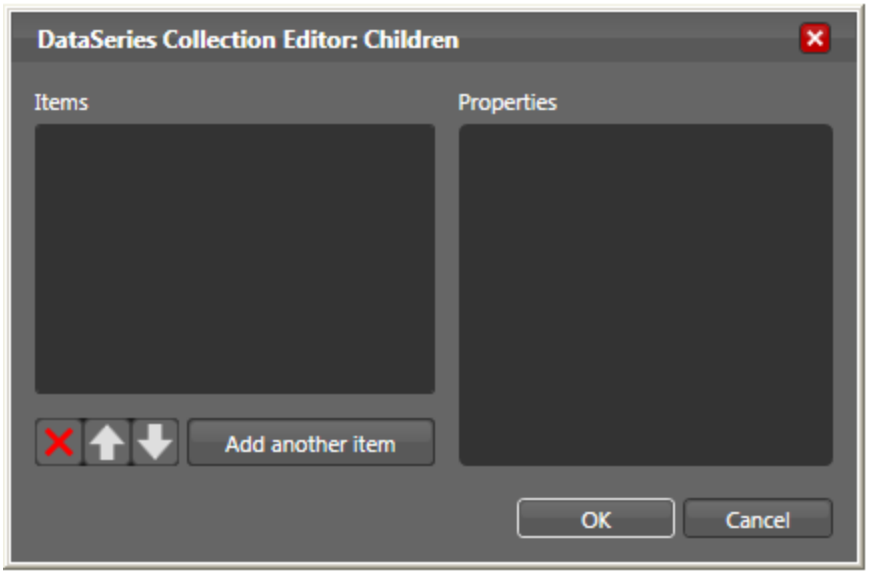
To add data to the chart in Blend using the Properties Window

1. Select the **C1Chart** control in the Window to make it active, then navigate to the Appearance tab in the **Chart Properties** window and set the **ChartType** property to Bar.
2. Navigate to the **Miscellaneous** tab in the **Chart Properties** window.
3. In the **Miscellaneous** tab, locate the Data (ChartData) and click on the **New** button. The chartdata object is added in the xaml code like the following:

```
<c1chart:C1Chart MinHeight="160" MinWidth="240" Content="C1Chart"
Margin="0,0,24,8">
                  <c1chart:C1Chart.Data>
                        <c1chart:ChartData/>
                  </c1chart:C1Chart.Data>
                  <c1chart:C1ChartLegend DockPanel.Dock="Right"/>
            </c1chart:C1Chart>
```

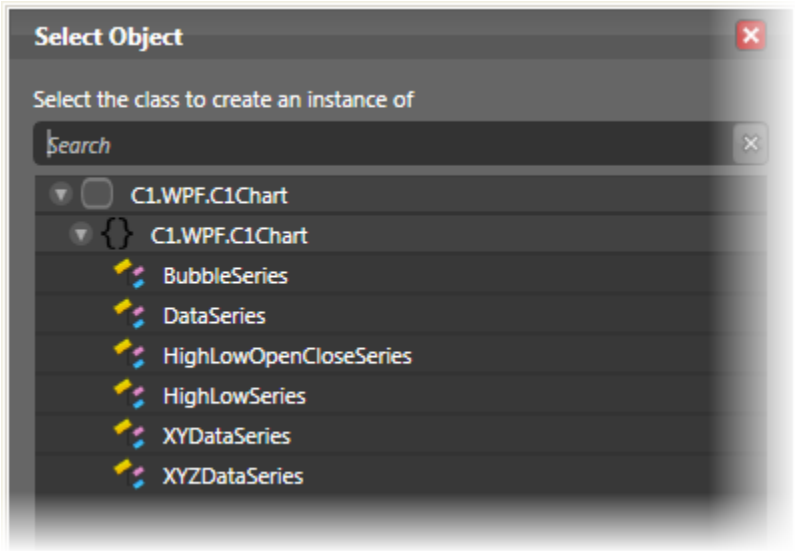
   The fake chart data is removed and the C1Chart control appears empty since you have not added any data to it yet.

4. Click on the arrow in the **C1Chart** properties window next to Data (ChartData) to expand the **ChartData** properties. Click on the ellipsis button next to the Children (Collection) property. The **DataSeries Collection Editor: Children** dialog box appears.



5. Click **Add another item** button to add a type of data series to the DataSeriesCollection.

   The **Select Object** dialog box appears.



In the **Select Object** dialog box you can choose one of the following chart objects (BubbleSeries, DataSeries, HighLowOpenCloseSeries, HighLowSeries, XYDataSeries, XYZDataSeries) depending on what type of chart you would like to create. To create a **Bar** chart, you would use the **DataSeries**. Once you select the **DataSeries** object, it is added to the **DataSeriesCollection**. To add multiple series you can click the **Add another item** button

6. Select **DataSeries** from the **Select Object** dialog box and click **OK**.

   The [0] DataSeries object is added to the Items panel.

7. Next we will add values that will represent the price of each product. Click on the ellipsis button next to the Values (Collection) to bring up the **Double Collection Editor: Values**.

8. Here you can add values. In this example we will click the **Add another item** button eight times to add eight values to our series.

9. Select the first array and double click the rectangular box in the Properties pane to enter the value, 80. Do the same for the remaining values and enter the following values respectively, 400, 20, 60, 150, 300, 130, and 500. Click **OK** when you are finished.

   The XAML code appears like the following:

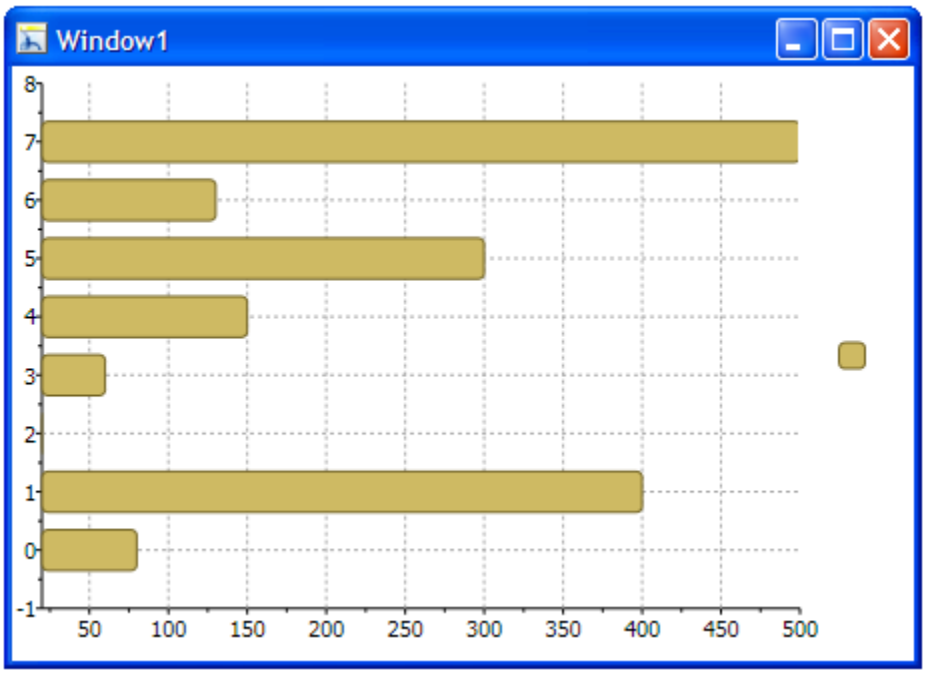
```
<c1chart:C1Chart.Data>
                        <c1chart:ChartData>
                              <c1chart:DataSeries Values="80 400 20 60 150
300 130 500"/>
                        </c1chart:ChartData>
                    </c1chart:C1Chart.Data>
```

10. Next we will add y values. Click on the ellipsis button next to the Values (Collection). The **Double Collection Editor:Values** appears.

11. Click on **Add another item** button four times to add four y-values to the first series. For the first y value enter 1, for the second y value enter 2, for the third y value enter 3, for the fourth y value enter 4. Once you are finished entering the y-values click **OK** to close the collection editor.

12. Navigate to the Appearance section in C1Chart's properties window and select the ChartType property and change it to Bar.

13. Run the project and you will have a simple **Bar** chart with one series that looks like the following:



14. Expand the **Data** tab and locate the ItemNames property. Enter the following names in this order: "Hand Mixer, Stand Mixer, Can Opener, Toaster, Blender, Food Processor, Slow Cooker, and Microwave. " Press the Enter key and then go back and make any modifications if needed in the XAML code. The XAML code should appear like the following:

```
<c1chart:ChartData.ItemNames>
                                        <x:Array Type="{x:Type
System:String}">
                                              <System:String>Hand
Mixer</System:String>
                                              <System:String>Stand
Mixer</System:String>
                                              <System:String>Can
Opener</System:String>

      <System:String>Toaster</System:String>

      <System:String>Blender</System:String>
                                              <System:String>Food
Processor</System:String>
                                              <System:String>Slow
Cooker</System:String>

      <System:String>Microwave</System:String>
                                        </x:Array>
                                  </c1chart:ChartData.ItemNames>
```

In the next step, Step 3 of 4: Format the Axes (page 38), you'll learn how to customize the axes using XAML code.

To add data to the chart programmatically in the code behind file

1. Create a new WPF project in Visual Studio.
2. Add the **C1Chart** control to Window1.

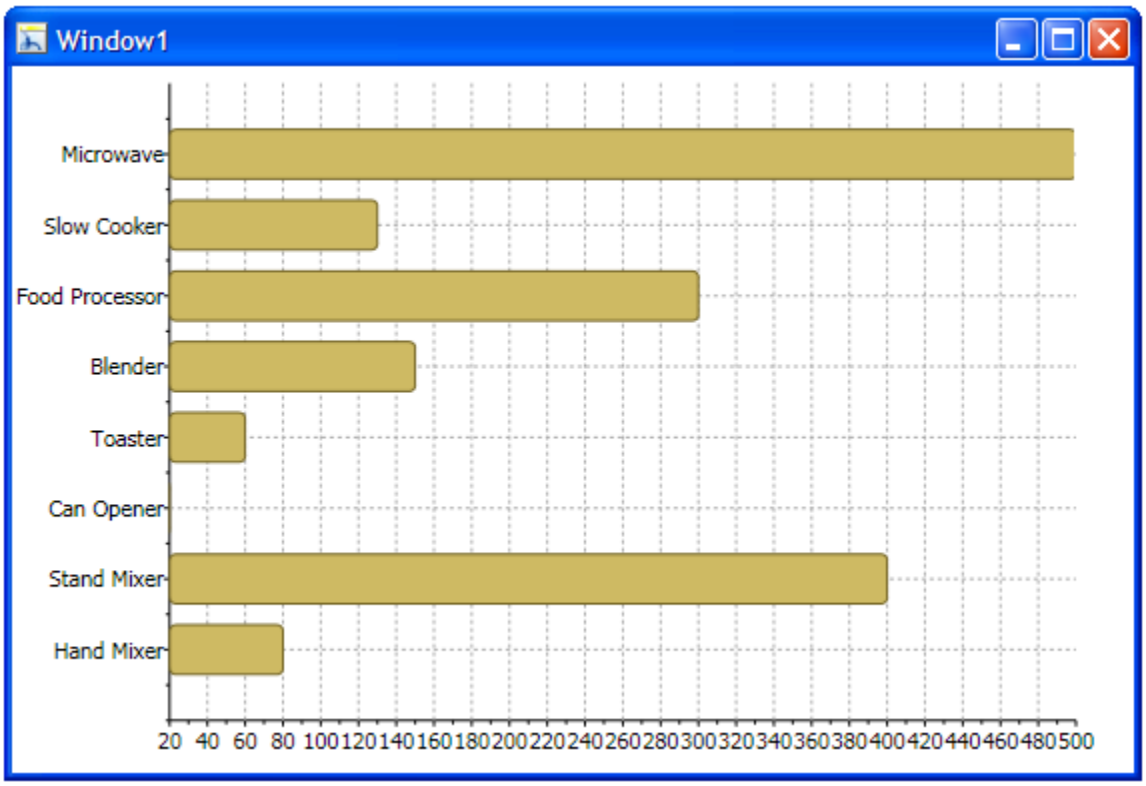
3. Right-click on Window1 and select **View Code** to open the editor.
4. Add the **C1.WPF.C1Chart** namespace directive
   - Visual Basic
   ```
   Imports C1.WPF.C1Chart
   ```
   - C#
   ```
   using C1.WPF.C1Chart;
   ```
5. Add the following code in the constructor Window1 class to create the Bar chart:
   - Visual Basic
   ```
   ' Clear previous data
   c1Chart1.Data.Children.Clear()

   ' Add Data
   Dim ProductNames As String() = {"Hand Mixer", "Stand Mixer", "Can
   Opener", "Toaster", "Blender", "Food Processor", _
   "Slow Cooker", "Microwave"}
   Dim PriceX As Integer() = {80, 400, 20, 60, 150, 300, _
   130, 500}

   ' create single series for product price
   Dim ds1 As New DataSeries()
   ds1.Label = "Price X"

   'set price data
   ds1.ValuesSource = PriceX

   ' add series to the chart
   c1Chart1.Data.Children.Add(ds1)

   ' add item names
   c1Chart1.Data.ItemNames = ProductNames

   ' Set chart type
   c1Chart1.ChartType = ChartType.Bar
   ```
   - C#
   ```
   // Clear previous data
      c1Chart1.Data.Children.Clear();

      // Add Data
      string[] ProductNames = { "Hand Mixer", "Stand Mixer", "Can Opener",
   "Toaster", "Blender", "Food Processor", "Slow Cooker", "Microwave" };
      int[] PriceX = { 80, 400, 20, 60, 150, 300, 130, 500 };

      // create single series for product price
      DataSeries ds1 = new DataSeries();
      ds1.Label = "Price X";
   ```

```
//set price data
ds1.ValuesSource = PriceX;

// add series to the chart
c1Chart1.Data.Children.Add(ds1);

// add item names
c1Chart1.Data.ItemNames = ProductNames;

// Set chart type
c1Chart1.ChartType = ChartType.Bar;
```

In the next step, Step 3 of 4: Format the Axes (page 38), you'll learn how to customize the axes programmatically

Run the program and observe:

The string values appear on the Y-Axis like the following:



Notice the X-Axis begins with 20 by default, in the next set of steps you'll customize the Axes.

You have successfully added data to the chart control. In the next step you'll format the axes.

Step 3 of 4: Format the Axes

In this step, you will add a ChartView object so you can customize the X-Axis.

To format the axes for Chart for WPF in Visual Studio using XAML:

1. Add the ChartView object so you can set titles for the X-Axis and Y-axis

   The ChartView object represents the area of the chart that contains data (including the axes). For more information on the chart axes, see Axes (page 98). The axis titles are UIElement objects rather than simple text. In this example we will use **TextBlock** elements to assign the text to the X-Axis and Y-Axis titles. Once we add the **TextBlock** element we can then format the text by changing its foreground color and aligning it to the center.

```
<c1chart:C1Chart >
      <c1chart:C1Chart.View>
        <c1chart:ChartView>
          <c1chart:ChartView.AxisX>
            <c1chart:Axis>
              <c1chart:Axis.Title>
                <TextBlock Text="Price" TextAlignment="Center"
Foreground="Crimson"/>
              </c1chart:Axis.Title>
            </c1chart:Axis>
          </c1chart:ChartView.AxisX>
          <c1chart:ChartView.AxisY>
            <c1chart:Axis>
              <c1chart:Axis.Title>
                <TextBlock Text="Kitchen Electronics"
TextAlignment="Center" Foreground="Crimson"/>
              </c1chart:Axis.Title>
            </c1chart:Axis>
          </c1chart:ChartView.AxisY>
        </c1chart:ChartView>
      </c1chart:C1Chart.View>
    </c1chart:C1Chart>
```

2. Configure the value of the X-Axis to start at zero and change the default AxisX.MajorUnit unit value from 50 to 20. Also set the AutoMin property to **False** so we can have the value begin at zero instead of the minimum data value. Your XAML code for the View object should now appear like the following:

```
<c1chart:C1Chart.View>
                <c1chart:ChartView>
                    <c1chart:ChartView.AxisX>
                        <c1chart:Axis Min="0" Max="500" MajorUnit="20"
AutoMin="False">
                            <c1chart:Axis.Title>
                                <TextBlock Text="Price"
TextAlignment="Center" Foreground="Crimson" />
                            </c1chart:Axis.Title>
                        </c1chart:Axis>
                    </c1chart:ChartView.AxisX>
                    <c1chart:ChartView.AxisY>
                        <c1chart:Axis>
                            <c1chart:Axis.Title>
                                <TextBlock Text="Kitchen Electronics"
TextAlignment="Center" Foreground="Crimson" />
                            </c1chart:Axis.Title>
                        </c1chart:Axis>
                    </c1chart:ChartView.AxisY>
                </c1chart:ChartView>
            </c1chart:C1Chart.View>
```

3. Within the <c1chart:Axis></c1chart:Axis> of the **ChartView.AxisX** object set the AnnoFormat to change the numeric x-values along the x-axis to currency and the AnnoAngle property to rotate the X-Axis annotation to 60 degrees counterclockwise.

```
<c1chart:Axis AnnoFormat="c" AnnoAngle="60"  />
```

4. Within the <c1chart:Axis></c1chart:Axis> of the **ChartView.AxisY** object set the Reversed property to **True** to reverse the direction of the Y-Axis.

   In the next step, Step 4 of 4: Adjust the Chart's Appearance (page 41), you'll learn how to customize the chart's appearance using XAML.

To format the axes for Chart for WPF programmatically in the code behind file

Add the following code in the constructor Window1 class to format the chart axes:

- Visual Basic

```
' set axes titles
   C1Chart1.View.AxisY.Title = New TextBlock(New Run("Kitchen
Electronics"))
   C1Chart1.View.AxisX.Title = New TextBlock(New Run("Price"))

   ' set axes bounds
   C1Chart1.View.AxisX.Min = 0
   C1Chart1.View.AxisX.Max = 500

   ' Financial formatting
   C1Chart1.View.AxisX.AnnoFormat = "c"

   ' axis annotation rotation
   C1Chart1.View.AxisX.AnnoAngle = "60"
```

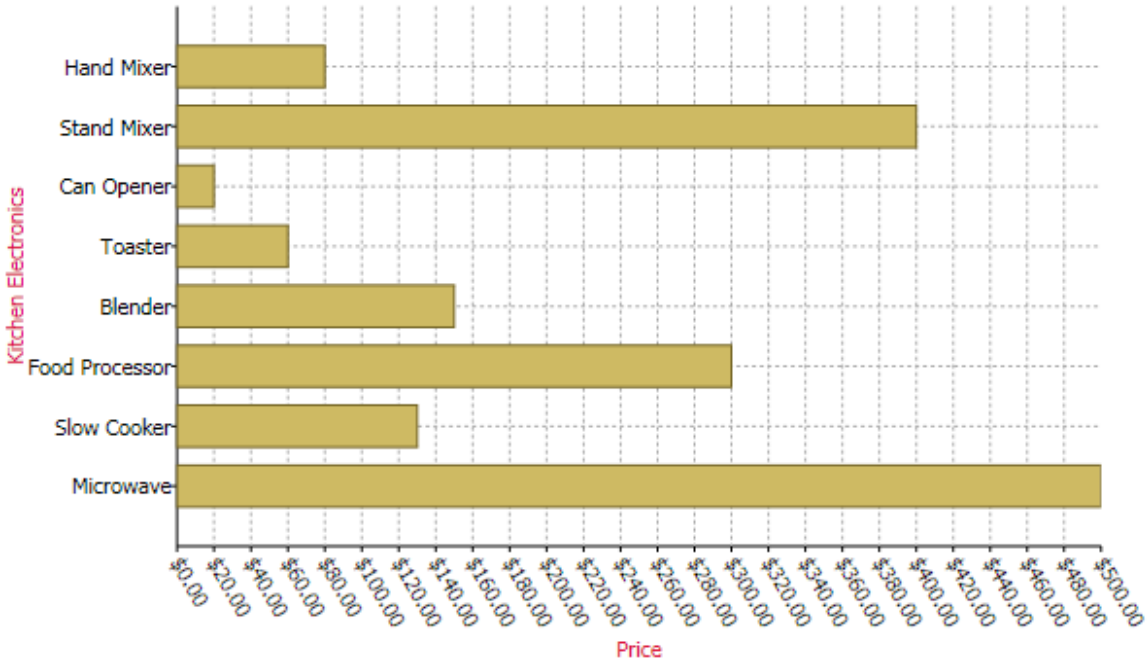
- C#

```
// set axes titles
   c1Chart1.View.AxisY.Title= new TextBlock() { Text = "Kitchen
Electronics" };
   c1Chart1.View.AxisX.Title = new TextBlock() { Text = "Price" };

   // set axes bounds
   c1Chart1.View.AxisX.Min = 0;
   c1Chart1.View.AxisX.Max = 500;

   // financial formatting
   c1Chart1.View.AxisX.AnnoFormat = "c";

   // axis annotation rotation
   c1Chart1.View.AxisX.AnnoAngle=60;
```

In the next step, Step 4 of 4: Adjust the Chart's Appearance (page 41), you'll learn how to customize the chart's appearance programmatically.

Run the program and observe:

The new format for the axis annotation is applied to the chart.



In the next step, you'll customize the chart's appearance using the one of the options from the Theme property.

Step 4 of 4: Adjust the Chart's Appearance

In this last step, you will adjust the chart's appearance using the Theme property.

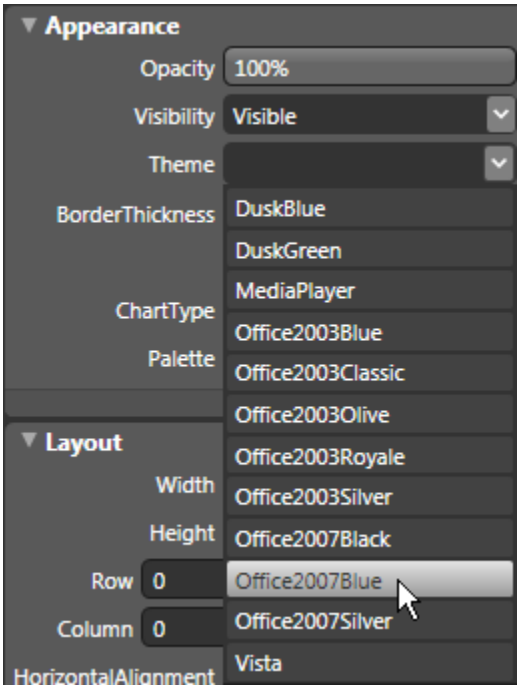
To set the chart's theme in Visual Studio using XAML:

To specifically define the **Office2007Blue** theme in your chart, add the following Theme XAML to the `<c1chart:C1Chart>` tag so that it appears similar to the following:

```
<c1chart:C1Chart Margin="0,0,8,8" MinHeight="160" MinWidth="240"
Content="C1Chart" ChartType="Bar" ThemeOffice2007Blue">
```

To set chart's theme in Blend using the Properties Window

1. Select the **C1Chart** control in Window1 to make it active.
2. Navigate to the C1Chart's **Appearance** group in the Properties window.
3. Click on the dropdown arrow next to the Theme property and select **Office2007Blue**.



To set chart's theme programmatically in the code behind file

To specifically define the **Office2007Blue** theme in your chart, add the following code your project:

- Visual Basic

```
C1Chart1.Theme = TryCast(C1Chart1.TryFindResource( _
   New ComponentResourceKey(GetType(C1Chart), " Office2007Blue")), _

   ResourceDictionary)
```

- C#

```
C1Chart1.Theme = c1Chart1.TryFindResource(

   new ComponentResourceKey(typeof(C1.WPF.C1Chart.C1Chart),

   "Office2007Blue")) as ResourceDictionary;
```

Run the program and observe:

The **Office 2007 Blue** theme is applied to the **C1Chart** control.

Congratulations! You've completed the **Chart for WPF** quick start and created a chart application, added data to the chart, set the axes bounds, formatted the axes annotation, and customized the appearance of the chart.

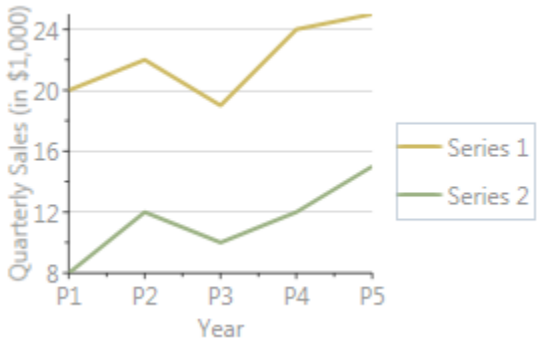
XAML Quick Reference

This section provides a few examples that show how to use the WPF **C1Chart** control with only XAML code.

EX: Set up a Basic Line Chart

The following XAML shows how to declare the C1Chart control, set the ChartType, Theme, and Palette properties to define the basic chart appearance. The XAML can be added inside the <Grid>…</Grid> tags.

The following chart is produced using the following XAML code:




```
<Window x:Class="WpfApplication1.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:c1="http://schemas.componentone.com/xaml/c1chart"
  Title="Window1" Height="300" Width="348" >
  <Grid>

<!--
  Declare the C1Chart control
  Set the ChartType, Theme, and Palette properties to define
  the basic Chart appearance
-->
    <c1:C1Chart
      Name="C1Chart1"
      ChartType="Line"
      Foreground="#a0000000"
      Background="#e0ffffff"
      Theme ="Vista"
      Palette ="Aspect" >

  <!--
    Define the chart View, which contains the chart axes.
  -->
      <c1:C1Chart.View>
        <c1:ChartView>

          <!-- Define the X axis (title, grid) -->
          <c1:ChartView.AxisX>
            <c1:Axis
              Title="Year"
              MajorGridStroke="Transparent"/>
          </c1:ChartView.AxisX>

          <!-- Define the Y axis (title, grid, annotation format) -->
          <c1:ChartView.AxisY>
            <c1:Axis
```

```
              Title="Quarterly Sales (in $1,000)"
              MajorGridStroke="#40000000"
              AnnoFormat="n0" />
          </c1:ChartView.AxisY>
        </c1:ChartView>
      </c1:C1Chart.View>

      <!--
        Define the chart Data, which contains the data series.
      -->
      <c1:C1Chart.Data>
        <c1:ChartData>

          <!-- ItemNames define the labels along the X axis -->
          <c1:ChartData.ItemNames>P1 P2 P3 P4
P5</c1:ChartData.ItemNames>

          <!--
             Each DataSeries specifies a label (shown in the legend)
             and the series data
          -->
          <c1:DataSeries Label="Series 1" RenderMode="Default"
Values="20 22 19 24 25" />
          <c1:DataSeries Label="Series 2" RenderMode="Default"
Values="8 12 10 12 15" />
        </c1:ChartData>
      </c1:C1Chart.Data>

<!--
  Add a ChartLegend, docked to the right of the chart, to display
  a legend containing the series and their styles.
-->
      <c1:C1ChartLegend DockPanel.Dock="Right" />
    </c1:C1Chart>
  </Grid>
</Window>
```

EX: Set up a Gantt Chart

To create a Gantt chart, use the following XAML code:

```
<c1chart:C1Chart Margin="0" Name="c1Chart1"
            xmlns:sys="clr-namespace:System;assembly=mscorlib">
  <c1chart:C1Chart.Resources>
    <x:Array x:Key="start" Type="sys:DateTime" >
      <sys:DateTime>2008-6-1</sys:DateTime>
      <sys:DateTime>2008-6-4</sys:DateTime>
      <sys:DateTime>2008-6-2</sys:DateTime>
    </x:Array>
    <x:Array x:Key="end" Type="sys:DateTime">
      <sys:DateTime>2008-6-10</sys:DateTime>
      <sys:DateTime>2008-6-12</sys:DateTime>
      <sys:DateTime>2008-6-15</sys:DateTime>
    </x:Array>
```

```
  </c1chart:C1Chart.Resources>
  <c1chart:C1Chart.Data>
    <c1chart:ChartData>
      <c1chart:ChartData.Renderer>
        <c1chart:Renderer2D Inverted="True" ColorScheme="Point"/>
      </c1chart:ChartData.Renderer>
      <c1chart:ChartData.ItemNames>Task1 Task2
Task3</c1chart:ChartData.ItemNames>
      <c1chart:HighLowSeries HighValuesSource="{StaticResource end}"
                             LowValuesSource="{StaticResource start}"/>
    </c1chart:ChartData>
  </c1chart:C1Chart.Data>
  <c1chart:C1Chart.View>
    <c1chart:ChartView>
      <c1chart:ChartView.AxisX>
        <c1chart:Axis IsTime="True" AnnoFormat="d"/>
      </c1chart:ChartView.AxisX>
    </c1chart:ChartView>
  </c1chart:C1Chart.View>
</c1chart:C1Chart>
```

EX: Create a Stacked Area Chart

To create a stacked area chart you should set the **C1Chart.ChartType** property instead of the **DataSeries.ChartType** like the following XAML example:

```
      <c1:C1Chart ChartType="AreaStacked" >
        <c1:C1Chart.Data>
          <c1:ChartData ItemNames="P1 P2 P3 P4 P5">
            <c1:DataSeries Label="Series 1" Values="20 22 19 24 25" />
            <c1:DataSeries Label="Series 2" Values="8 12 10 12 15" />
          </c1:ChartData>
        </c1:C1Chart.Data>
        <c1:C1ChartLegend DockPanel.Dock="Right" />
      </c1:C1Chart>
```

Chart for WPF Top Tips

The following top tips for **Chart for WPF** will help you when you use the C1Chart control.

Tip 1: Use the BeginUpdate()/EndUpdate methods to improve performance

When performing a massive update of the chart properties or the data values, put the update code inside a **BeginUpdate()/EndUpdate()** block.

This improves performance since the redrawing occurs only once after a call of the **EndUpdate()** method.

For example:

- Visual Basic

```
' start update
C1Chart1.BeginUpdate()

Dim nser As Integer = 10, npts As Integer = 100
For iser As Integer = 1 To nser

  ' create data arrays
  Dim x(npts - 1) As Double, y(npts - 1) As Double
  For ipt As Integer = 0 To npts - 1
    x(ipt) = ipt
    y(ipt) = (1 + 0.05 * iser) * Math.Sin(0.1 * ipt + 0.1 * iser)
  Next

  ' create data series and add it to the chart
  Dim ds = New XYDataSeries()
  ds.XValuesSource = x
  ds.ValuesSource = y
  C1Chart1.Data.Children.Add(ds)
Next

' set chart type
C1Chart1.ChartType = ChartType.Line

' finish update
C1Chart1.EndUpdate()
```

- C#

```
// start update
c1Chart1.BeginUpdate();

int nser = 10, npts = 100;
for (int iser = 0; iser < nser; iser++)
{
  // create data arrays
  double[] x = new double[npts], y = new double[npts];
  for (int ipt = 0; ipt < npts; ipt++)
  {
    x[ipt] = ipt;
    y[ipt] = (1 + 0.05 * iser) * Math.Sin(0.1 * ipt + 0.1 * iser);
  }
```

```
    // create data series and add it to the chart
    XYDataSeries ds = new XYDataSeries();
    ds.XValuesSource = x; ds.ValuesSource = y;
    c1Chart1.Data.Children.Add(ds);
  }

  // set chart type
  c1Chart1.ChartType = ChartType.Line;

  // finish update
  c1Chart1.EndUpdate();
```

Tip 2: Use the line or area chart type for large data arrays

The line and area charts provide the best performance when you have a lots of data values.

To get better performance, enable built-in optimization for large data by setting the attached property, **LineAreaOptions.OptimizationRadius**. For example:

- Visual Basic

```
LineAreaOptions.SetOptimizationRadius(C1Chart1, 1.0)
```

- C#

```
LineAreaOptions.SetOptimizationRadius(c1Chart1, 1.0);
```

It's recommended you use small values 1.0 - 2.0 as radius. A larger value may affect the accuracy of the plot.

Tip 3: Update the appearance and behavior of a plot element using the DataSeries.PlotElementLoaded event

When any plot element (bar,column,pie, etc) is loaded it fires the PlotElementLoaded event. During this event you have access to the plot element properties as well as to the corresponding data point.

The following code sets the colors of points depending on its y-value. For example:

- Visual Basic

```
' create data arrays
  Dim npts As Integer = 100
  Dim x(npts - 1) As Double, y(npts - 1) As Double
  For ipt As Integer = 0 To npts - 1
    x(ipt) = ipt
    y(ipt) = Math.Sin(0.1 * ipt)
  Next

  ' create data series
  Dim ds = New XYDataSeries()
  ds.XValuesSource = x
  ds.ValuesSource = y

  ' set event handler
  AddHandler ds.PlotElementLoaded, AddressOf PlotElement_Loaded

  ' add data series to the chart
  C1Chart1.Data.Children.Add(ds)

  ' set chart type
```

```
  C1Chart1.ChartType = ChartType.LineSymbols

...

  ' event handler
  Sub PlotElement_Loaded(ByVal sender As Object, ByVal args As EventArgs)
    Dim pe = CType(sender, PlotElement)
    If Not TypeOf pe Is Lines Then
      Dim dp As DataPoint = pe.DataPoint

      ' normalized y-value(from 0 to 1)
      Dim nval As Double = 0.5 * (dp.Value + 1)

      ' fill from blue(-1) to red(+1)
      pe.Fill = New SolidColorBrush(Color.FromRgb(CByte(255 * nval), _
       0, CByte(255 * (1 - nval))))
    End If
  End Sub
```

- *C#*

```
// create data arrays
  int npts = 100;
  double[] x = new double[npts], y = new double[npts];
  for (int ipt = 0; ipt < npts; ipt++)
  {
    x[ipt] = ipt;
    y[ipt] = Math.Sin(0.1 * ipt);
  }

  // create data series
  XYDataSeries ds = new XYDataSeries();
  ds.XValuesSource = x; ds.ValuesSource = y;

  // set event handler
  ds.PlotElementLoaded += (s, e) =>
    {
      PlotElement pe = (PlotElement)s;
      if (!(pe is Lines)) // skip lines
      {
        DataPoint dp = pe.DataPoint;

        // normalized y-value(from 0 to 1)
        double nval = 0.5*(dp.Value + 1);

        // fill from blue(-1) to red(+1)
        pe.Fill = new SolidColorBrush(
          Color.FromRgb((byte)(255 * nval), 0, (byte)(255 * (1-nval))));
      }
    };

  // add data series to the chart
  c1Chart1.Data.Children.Add(ds);
```

```
// set chart type
c1Chart1.ChartType = ChartType.LineSymbols;
```

Tip 4: Data point labels and tooltips

To create a data point label or tooltip, you should set the data template for the PointLabelTemplate or PointTooltipTemplate property.

The following sample code shows the index for each data point.

XAML:

```
<c1chart:C1Chart Name="c1Chart1" ChartType="XYPlot">
  <c1chart:C1Chart.Data>
    <c1chart:ChartData>
      <!-- source collection -->
      <c1chart:ChartData.ItemsSource>
        <PointCollection>
          <Point X="1" Y="1" />
          <Point X="2" Y="2" />
          <Point X="3" Y="3" />
          <Point X="4" Y="2" />
          <Point X="5" Y="1" />
        </PointCollection>
      </c1chart:ChartData.ItemsSource>

      <c1chart:XYDataSeries SymbolSize="16,16"
        XValueBinding="{Binding X}" ValueBinding="{Binding Y}">
        <c1chart:XYDataSeries.PointLabelTemplate>
          <DataTemplate>
            <!-- display point index at the center of point symbol  -->
            <TextBlock
c1chart:PlotElement.LabelAlignment="MiddleCenter"
                       Text="{Binding PointIndex}" />
          </DataTemplate>
        </c1chart:XYDataSeries.PointLabelTemplate>
      </c1chart:XYDataSeries>
    </c1chart:ChartData>
  </c1chart:C1Chart.Data>
</c1chart:C1Chart>
```

The data context of element created from the template is set to the instance of DataPoint class which contains information about the corresponding data point.

Tip 5: Save chart as image

The following method saves chart image as png-file.

- Visual Basic

```
Sub Using stm = System.IO.File.Create(fileName)
    c1Chart1.SaveImage(stm, ImageFormat.Png)
End Using
```

- C#

```
using (var stm = System.IO.File.Create(fileName))
   {
    c1Chart1.SaveImage(stm, ImageFormat.Png);
   }
```

Tip 6: Printing chart

The following code prints the specified chart on the default printer with the default settings. For example:

- Visual Basic

```
Dim pd = New PrintDialog()
  pd.PrintVisual(C1Chart1, "chart")
```

- C#

```
new PrintDialog().PrintVisual(c1Chart1, "chart");
```

Tip 7: Mixing Cartesian chart types

You can easily mix different chart types on the same Cartesian plot using the ChartType property.

The following code creates three data series: the first is area, the second is step, and the third has the default chart type (line).

- Visual Basic

```
Dim nser As Integer = 3, npts As Integer = 25
  For iser As Integer = 1 To nser

    ' create data arrays
    Dim x(npts - 1) As Double, y(npts - 1) As Double
    For ipt As Integer = 0 To npts - 1
      x(ipt) = ipt
      y(ipt) = (1 + 0.05 * iser) * Math.Sin(0.1 * ipt + 0.1 * iser)
    Next

    ' create data series and add it to the chart
    Dim ds = New XYDataSeries()
    ds.XValuesSource = x
    ds.ValuesSource = y
    C1Chart1.Data.Children.Add(ds)
  Next

  'default chart type
  C1Chart1.ChartType = ChartType.Line

  ' 1st series
  C1Chart1.Data.Children(0).ChartType = ChartType.Area

  ' 2nd series
  C1Chart1.Data.Children(1).ChartType = ChartType.Step
```

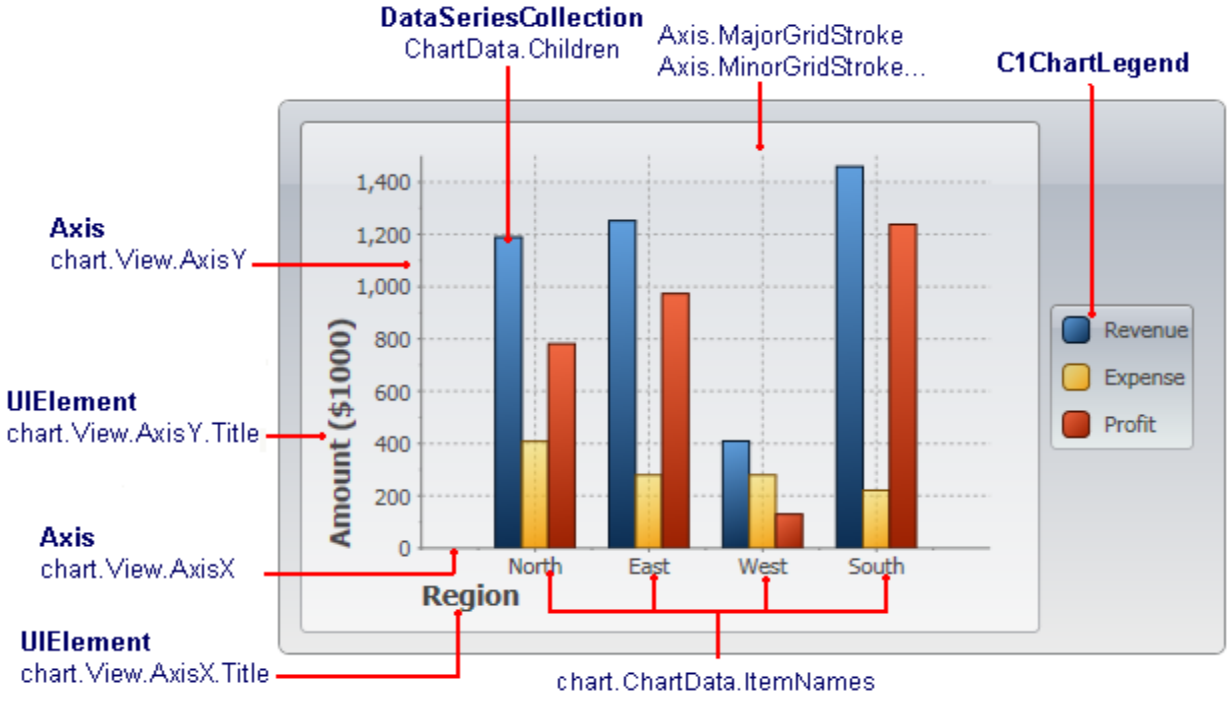
- C#

```
int nser = 3, npts = 25;
  for (int iser = 0; iser < nser; iser++)
  {

    // create data arrays
    double[] x = new double[npts], y = new double[npts];
    for (int ipt = 0; ipt < npts; ipt++)
    {
      x[ipt] = ipt;
      y[ipt] = (1 + 0.05 * iser) * Math.Sin(0.1 * ipt + 0.1 * iser);
    }
```

```
    // create data series and add it to the chart
    XYDataSeries ds = new XYDataSeries();
    ds.XValuesSource = x; ds.ValuesSource = y;
    c1Chart1.Data.Children.Add(ds);
  }

  //default chart type
  c1Chart1.ChartType = ChartType.Line;

  // 1st series
  c1Chart1.Data.Children[0].ChartType = ChartType.Area;

  // 2nd series
  c1Chart1.Data.Children[1].ChartType = ChartType.Step;
```

C1Chart Concepts and Main Properties

In order to create and format charts using the **C1Chart** control, it is useful to understand how the main properties map into chart elements. The diagram below illustrates this:



The steps involved in creating a typical chart are:

1. Choose the chart type (ChartType property)
   C1Chart supports about 30 chart types, including Bar, Column, Line, Area, Pie, Radial, Polar, Candle,

and several others. The best chart type depends largely on the nature of the data, and will be discussed later.

2. Set up the axes (AxisX and **chart.View.AxisY** properties)
   Setting up the axes typically involves specifying the axis title, major and minor intervals for the tick marks, content and format for the labels to show next to the tick marks.
3. Add one or more data series (**chart.Data.Children** collection)
   This step involves creating and populating one **DataSeries** object for each series on the chart, then adding the object to the **chart.Data.Children** collection. If your data contains only one numeric value per point (Y coordinate), use regular **DataSeries** objects. If the data contains two numeric values per point (X and Y coordinates), then use **XYDataSeries** objects instead.
4. Adjust the chart's appearance using the **Theme** and **Palette** properties.
   The **Theme** property allows you to select one of over 10 built-in sets of properties that control the appearance of the overall chart. The **Palette** property allows you to select one of over 20 built-in color palettes used to specify colors for the data series. Together, these two properties provide about 200 options to create professionally-looking charts with little effort.

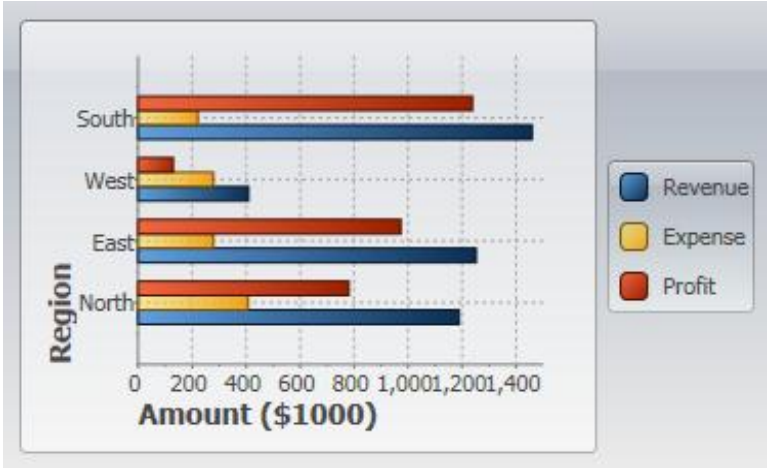
Common Usage for Basic 2D Charts

This chapter describes the common usage of the basic chart types such as Bar, Pie, and X-Y Plot charts. It also provides a sample code for each chart type. The samples are simple and concise, and focus on the main aspects of each common chart type. The distribution package includes a lot of sophisticated samples that show details and advanced features not discussed in this quick walkthrough.

This section describes how to create basic chart types, including the selection of chart type, adding the data, formatting, and adding titles to the chart axes.

Simple Charts

The simplest charts are those in which each data point has a single numeric value associated with it. A typical example would be a chart showing sales data for different regions, similar to the following chart:



Before we can create any charts, we need to generate the data that will be shown as a chart. Here is some code to create the data we need.

Note: There is nothing chart-specific in this code, this is just some generic data. We will use this data to create the Time Series and XY charts as well in the next topics.

```
// Simple class to hold dummy sales data
public class SalesRecord
{
  // Properties
    public string Region  { get; set; }
    public string Product { get; set; }
    public DateTime Date  { get; set; }
    public double Revenue { get; set; }
    public double Expense { get; set; }
    public double Profit  { get { return Revenue - Expense; } }
  // Constructor 1
    public SalesRecord(string region, double revenue, double expense)
    {
      Region = region;
      Revenue = revenue;
      Expense = expense;
    }
  // Constructor 2
    public SalesRecord(DateTime month, string product,    double revenue,
double expense)
    {
      Date = month;
      Product = product;
      Revenue = revenue;
      Expense = expense;
    }
}
// Return a list with one SalesRecord for each region
List<SalesRecord> GetSalesPerRegionData()
{
  var data = new List<SalesRecord>();
  Random rnd = new Random(0);
  foreach (string region in "North,East,West,South".Split(','))
  {
    data.Add(new SalesRecord(region, 100 + rnd.Next(1500),
rnd.Next(500)));
  }
  return data;
}
// Return a list with one SalesRecord for each product,// Over a period of 12
months
List<SalesRecord> GetSalesPerMonthData()

{
  var data = new List<SalesRecord>();
  Random rnd = new Random(0);
  string[] products = new string[] {"Widgets", "Gadgets", "Sprockets"
};
  for (int i = 0; i < 12; i++)
```

```
  {
    foreach (string product in products)
    {
      data.Add(new SalesRecord(
        DateTime.Today.AddMonths(i - 24),
        product,
        rnd.NextDouble() * 1000 * i,
        rnd.NextDouble() * 1000 * i));
      }
    }
    return data;
     }
}
```

Note that the **SalesData** class is public. This is required for data-binding.

We will follow the following four main steps in creating a chart:

Step 1) Choose the chart type:

The following code clears any existing series, then sets the chart type:

```
public Window1()
{
    InitializeComponent();
// Clear current chart
     c1Chart.Reset(true);
// Set chart type
   c1Chart.ChartType = ChartType.Bar;
}
```

Step 2) Set up the axes:

We will start by obtaining references to both axes. In most charts, the horizontal axis (X) displays labels associated with each point, and the vertical axis (Y) displays the values. The exception is the Bar chart type, which displays horizontal bars. For this chart type, the labels are displayed on the Y axis and the values on the X:

Next we will assign titles to the axes. The axis titles are **UIElement** objects rather than simple text. This means you have complete flexibility over the format of the titles. In fact, you could use complex elements with buttons, tables, or images for the axis titles. In this case, we will use simple **TextBlock** elements created by a **CreateTextBlock** method described later.

We will also configure the value axis to start at zero, and to display the annotations next to the tick marks using thousand separators:

```
  // configure label axis
  labelAxis.Title = CreateTextBlock("Region", 14, FontWeights.Bold);

  // configure value axis
_c1Chart.View.AxisX.Title = CreateTextBlock("Amount ($1000)", 14,
FontWeights.Bold);
  c1Chart.View.AxisX.AutoMin = false;
  c1Chart.View.AxisX.Min = 0;
  c1Chart.View.AxisX.MajorUnit = 200;
  c1Chart.View.AxisX.AnnoFormat = "#,##0 ";
```

Step 3) Add one or more data series

We start this step by retrieving the data using the method listed earlier:

```
  // get the data
  var data = GetSalesPerRegionData();
```

Next, we want to display the regions along the label axis. To do this, we will use a Linq statement that retrieves the **Region** property for each record. The result is then converted to an array and assigned to the ItemNames property.

```
// Show regions along label axis
  c1Chart.ChartData.ItemNames = (from r in data select
r.Region).ToArray();
```

Note how the use of Linq makes the code direct and concise. Things are made even simpler because our sample data contains only one record per region. In a more realistic scenario, there would be several records per region, and we would use a more complex Linq statement to group the data per region.

Now we are ready to create the actual **DataSeries** objects that will be added to the chart. We will create three series: "Revenue", "Expenses", and "Profit":

```
  // Add Revenue series
  var ds = new DataSeries();
  ds.Label = "Revenue";
  ds.ValuesSource = (from r in data select r.Revenue).ToArray();
c1Chart.Data.Children.Add(ds);
  // Add Expense series
  ds = new DataSeries();
  ds.Label = "Expense";
  ds.ValuesSource = (from r in data select r.Expense).ToArray();
  c1Chart.ChartData.Children.Add(ds);
  // Add Profit series
  ds = new DataSeries();
  ds.Label = "Profit";
  ds.ValuesSource = (from r in data select r.Profit).ToArray();
  c1Chart.Data.Children.Add(ds);
```

For each series, the code creates a new **DataSeries** object, then sets its **Label** property. The label is optional; if provided, it will be displayed in any **C1ChartLegend** objects associated with this chart. Next, a Linq statement is used to retrieve the values from the data source. The result is assigned to the **ValuesSource** property of the data series object. Finally, the data series is added to the chart's Children collection.

Once again, note how the use of Linq makes the code concise and natural.

Step 4) Adjust the chart's appearance

We will use the Theme property to quickly configure the chart appearance:

```
// Set theme
c1Chart.Theme = _c1Chart.TryFindResource(new
ComponentResourceKey(typeof(C1.WPF.C1Chart.C1Chart),"Office2007Black"))
as ResourceDictionary;}
```

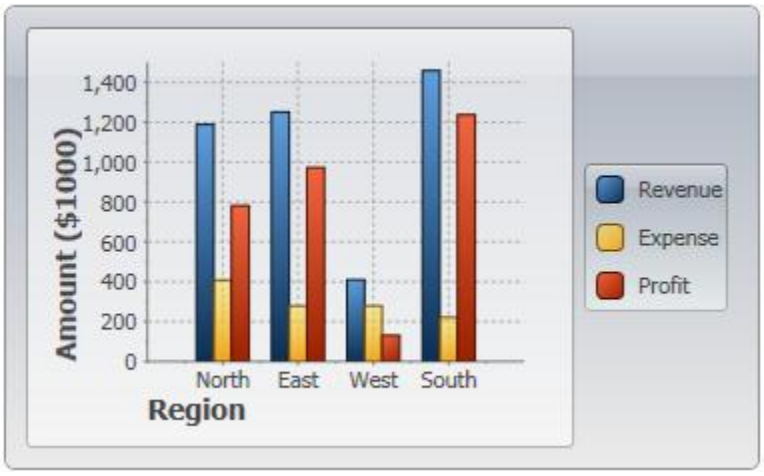
Recall that we used a **CreateTextBlock** helper method when setting up the axes. Here is the method definition:

```
TextBlock CreateTextBlock(string text, double fontSize, FontWeight
fontWeight)
{
  var tb = new TextBlock();
  tb.Text = text;
  tb.FontSize = fontSize;
  tb.FontWeight = fontWeight;
  return tb;
}
```

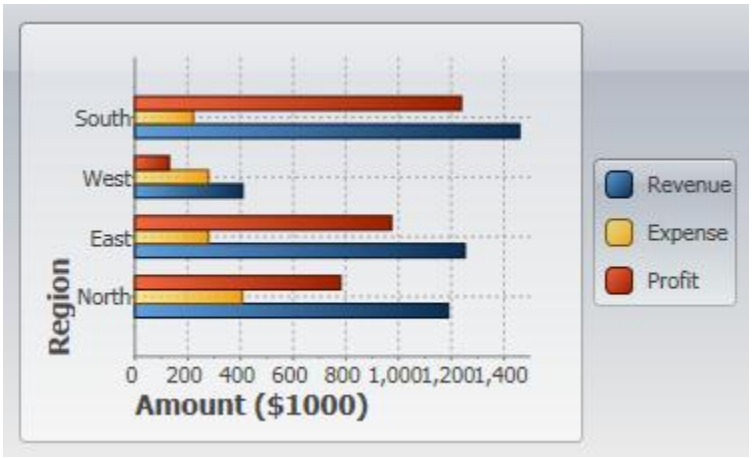
This concludes the code that generates simple value charts. You can test it by invoking the changing to value of the ChartType property to any of the remaining simple chart type values: Bar, AreaStacked, and Pie to create charts of

different types. Note, if you change the ChartType to Column, you will need display the labels on the Y-Axis so you will use AxisY. The result should be similar to the images below:

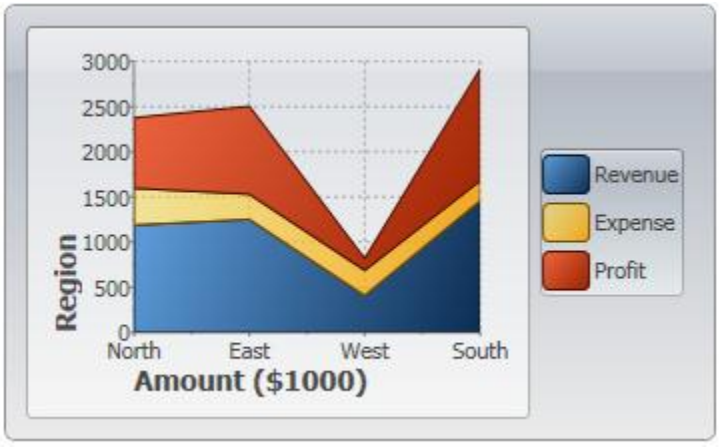
ChartType.Column



ChartType.Bar



ChartType.AreaStacked



ChartType.Pie


```
<c1chart:C1ChartLegend DockPanel.Dock="Right" />
```

Note: By default the chart displays a legend describing the series. To remove the C1ChartLegend, delete the following XAML code:

Time-Series Charts

Time-series charts display time along the X-axis. This is a very common type of chart, used to show how values change as time passes.

Most time-series charts show constant time intervals (yearly, monthly, weekly, daily). In this case, the time-series chart is essentially identical to a simple value type chart like the one described above. The only difference is that instead of showing categories along the X axis, the chart will show dates or times. (If the time intervals are not constant, then the chart becomes an XY chart, described in the next section.)

We will now walk through the creation of some time-series charts.

Step 1) Choose the chart type:

The code clears any existing series, then sets the chart type:

```
public Window1()
   {
       InitializeComponent();
// Clear current chart
       c1Chart.Reset(true);
// Set chart type
      c1Chart.ChartType = ChartType.Column;
   }
```

Step 2) Set up the axes:

We will start by obtaining references to both axes, as in the previous sample. Recall that the **Bar** chart type uses reversed axes (values are displayed on the Y axis):

```
//Get axes
Axis valueAxis = c1Chart.View.AxisY;
Axis labelAxis = c1Chart.View.AxisX;
```

```
    if (c1Chart.ChartType == ChartType.Bar)
    {
        valueAxis = _c1Chart.View.AxisX;
        labelAxis = _c1Chart.View.AxisY;
     }
```

Next we will assign titles to the axes. The axis titles are **UIElement** objects rather than simple text. We will set up the axis titles using the **CreateTextBlock** method, the same way we did before. We will also set up the annotation format, minimum value, and major unit. The only difference is we will use a larger interval for the tick marks between values:

```
// configure label axis
labelAxis.Title = CreateTextBlock("Date", 14, FontWeights.Bold);
labelAxis.AnnoFormat = "MMM-yy";

// configure value axis
valueAxis.Title = CreateTextBlock("Amount ($1000)", 14,
FontWeights.Bold);
valueAxis.AnnoFormat = "#,##0 ";
valueAxis.MajorUnit = 1000;
valueAxis.AutoMin = false;
valueAxis.Min = 0;
```

Step 3) Add one or more data series

This time, we will use the second data-provider method defined earlier:

```
// get the data
var data = GetSalesPerMonthData();
```

Next, we want to display the dates along the label axis. To do this, we will use a Linq statement that retrieves the distinct **Date** values in our data records. The result is then converted to an array and assigned to the ItemsSource property of the label axis.

```
c1Chart.ChartData.ItemNames = (from r in data select
r.Date.ToString("MMM-yy")).Distinct().ToArray();
```

Note that we used the **Distinct** Linq operator to remove duplicate date values. That is necessary because our data contains one record per product for each date.

Now we are ready to create the actual **DataSeries** objects that will be added to the chart. Each series will show the revenue for a given product. This can be done with a Linq statement that is slightly more elaborate than what we used before, but provides a good practical example of the power provided by Linq:

```
// add one series (revenue) per product
var products = (from p in data select p.Product).Distinct();
foreach (string product in products)
{
  var ds = new DataSeries();
  ds.Label = product;
  ds.ValuesSource = (
    from r in data
      where r.Product == product
      select r.Revenue).ToArray();
     c1Chart.ChartData.Children.Add(ds);
}
```

The code starts by building a list of products in the data source. Next, it creates one **DataSeries** for each product. The label of the data series is simply the product name. The actual data is obtained by filtering the records that belong to the current product and retrieving their **Revenue** property. The result is assigned to the **ValuesSource** property of the data series as before.

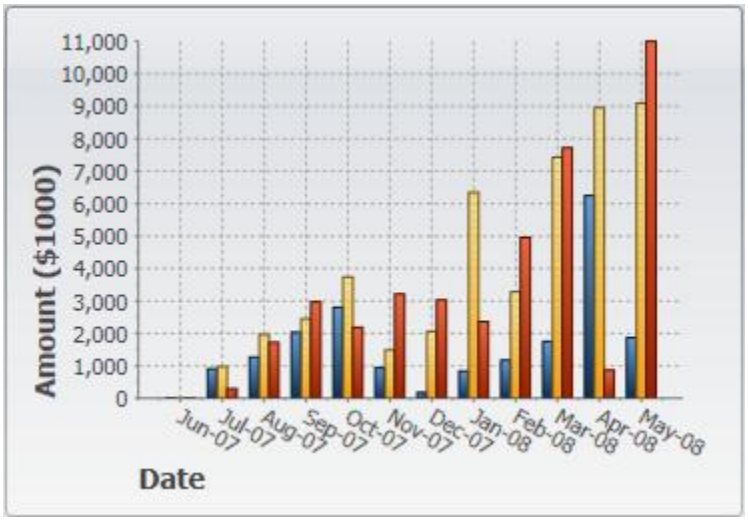
Step 4) Adjust the chart's appearance

Once again, we will finish by setting the Theme and Palette properties to quickly configure the chart appearance:

```
c1Chart.Theme = c1Chart.TryFindResource(
          new ComponentResourceKey(typeof(C1.WPF.C1Chart.C1Chart),
          "Office2007Black")) as ResourceDictionary;}
```

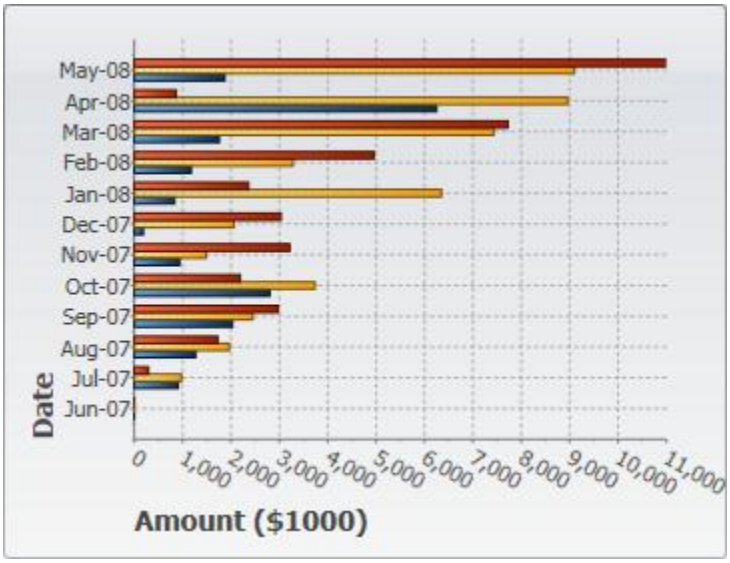
This concludes the code that generates our time-series charts. You can test it by running it and changing the ChartType property to **Bar**, **Column**, **AreaStacked**, or **Pie** to create charts of different types. The result should be similar to the images below:

ChartType.Column



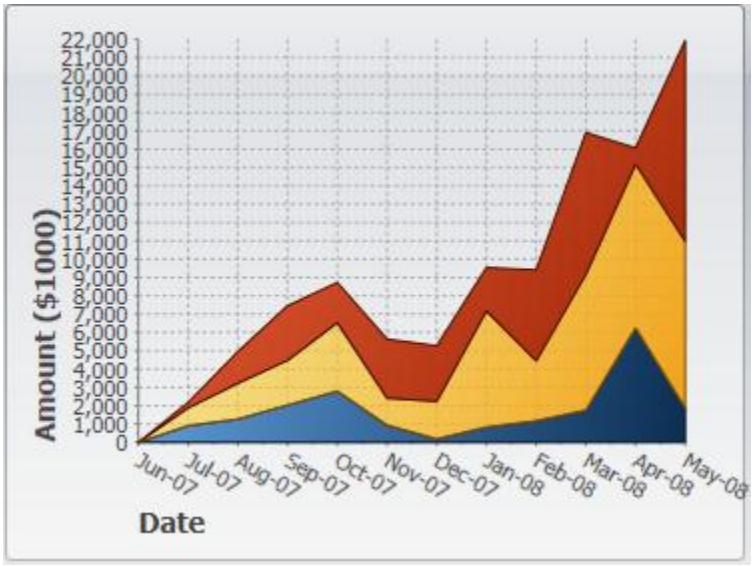
Note: The AnnoAngle property was set to "30" to make room for the Axis X labels in the images above.

ChartType.Bar



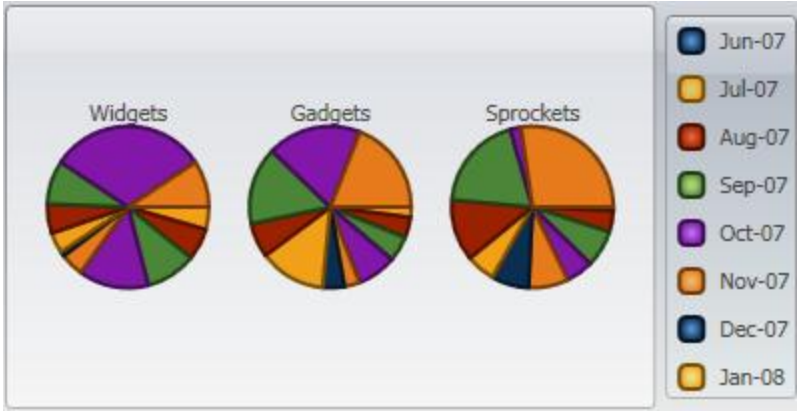
Note: The AnnoAngle property was set to "30" to make room for the Axis Y labels in the images above.

ChartType.AreaStacked



Note: The AnnoAngle property was set to "30" to make room for the Axis X labels in the image above.

ChartType.Pie



You would probably never display a time-series chart as a pie. As you can see from the image, the pie chart completely masks the growth trend that is clearly visible in the other charts.

XY Charts

XY charts (also known as scatter plots) are used to show relationships between variables. Unlike the charts we introduced so far, in XY charts each point has two numeric values. By plotting one of the values against the X axis and one against the Y axis, the charts show the effect of one variable on the other.

We will continue our **C1Chart** tour using the same data we created earlier, but this time we will create XY charts that show the relationship between revenues from two products. For example, we might want to determine whether high **Widget** revenues are linked to high **Gadgets** revenues (perhaps the products work well together), or whether high **Widget** revenues are linked to low **Gadgets** revenues (perhaps people who buy one of the products don't really need the other).

To do this, we will follow the same steps as before. The main differences are that this time we will add **XYDataSeries** objects to the chart's **Data.Children** collection instead of the simpler **DataSeries** objects. The Linq statement used to obtain the data is also a little more refined and interesting.

Step 1) Choose the chart type:

The code clears any existing series, then sets the chart type:

```
public Window1()
{
    InitializeComponent();

// Clear current chart
    c1Chart.Reset(true);

// Set chart type
    c1Chart.ChartType = ChartType.XYPlot;
```

Step 2) Set up the axes:

Since we're now creating XY series, we have two value axes (before we had a label axis and a value axis). We will attach titles and formats to both axes as we did before. We will also set the scale and annotation format as before. We will use the **AnnoAngle** property to rotate the annotation labels along the X axis so they don't overlap:

```
// get axes
    var yAxis = _c1Chart.View.AxisY;
    var xAxis = _c1Chart.View.AxisX;

// configure Y axis
```

```
  yAxis.Title = CreateTextBlock("Widget Revenues", 14,
FontWeights.Bold);
  yAxis.AnnoFormat = "#,##0 ";
  yAxis.AutoMin = false;
  yAxis.Min = 0;
  yAxis.MajorUnit = 2000;
  yAxis.AnnoAngle = 0;
```

```
// configure X axis
  xAxis.Title = CreateTextBlock("Gadget Revenues", 14,
FontWeights.Bold);
  xAxis.AnnoFormat = "#,##0 ";
  xAxis.AutoMin = false;
  xAxis.Min = 0;
  xAxis.MajorUnit = 2000;
  xAxis.AnnoAngle = -90; // rotate annotations
```

Step 3) Add one or more data series

Once again, we will use the second data-provider method defined earlier:

```
// get the data
  var data = GetSalesPerMonthData();
```

Next, we need to obtain XY pairs that correspond to the total revenues for **Widgets** and **Gadgets** at each date. We can use Linq to obtain this information directly from our data:

```
// group data by sales date
  var dataGrouped = from r in data
    group r by r.Date into g
    select new
    {
      Date = g.Key, // group by date
      Widgets = (from rp in g // add Widget revenues
        where rp.Product == "Widgets"
        select g.Sum(p => rp.Revenue)).Single(),
      Gadgets = (from rp in g // add Gadget revenues
        where rp.Product == "Gadgets"
        select g.Sum(p => rp.Revenue)).Single(),
    };
```

```
// sort data by widget sales
    var dataSorted = from r in dataGrouped
      orderby r.Gadgets
      select r;
```

The first Linq query starts by grouping the data by **Date**. Then, for each group it creates a record containing the **Date** and the sum of revenues within that date for each of the products we are interested in. The result is a list of objects with three properties: **Date**, **Widgets**, and **Gadgets**. This type of data grouping and aggregation is a powerful feature of Linq.

The second Linq query simply sorts the data by **Gadget** revenue. These are the values that will be plotted on the X axis, and we want them to be in ascending order. Plotting unsorted values would look fine if we displayed only symbols (**ChartType = XYPlot**), but it would look messy if we chose other chart types such as **Line** or **Area**.

Once the data has been properly grouped, summarized, and sorted, all we need to do is create one single data series, and assign one set of values to the ValuesSource property and the to the XValuesSource property:

```
// create the new XYDataSeries
  var ds = new XYDataSeries();

// set series label (displayed in a C1ChartLegend)
  ds.Label = "Revenue:\r\nWidgets vs Gadgets";

// populate Y values
  ds.ValuesSource = (
    from r in dataSorted
    select r.Widgets).ToArray();

// populate X values
  ds.XValuesSource = (
    from r in dataSorted
    select r.Gadgets).ToArray();

// add the series to the chart
  c1Chart.ChartData.Children.Add(ds);
```

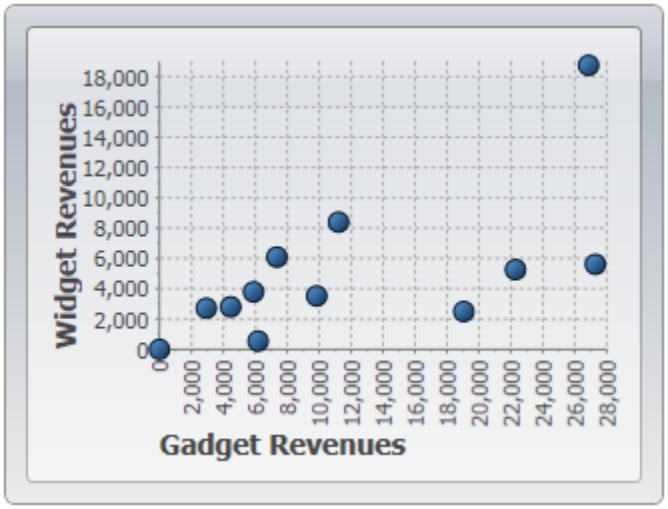
Step 4) Adjust the chart's appearance

Once again, we will finish by setting the Theme property to quickly configure the chart appearance:

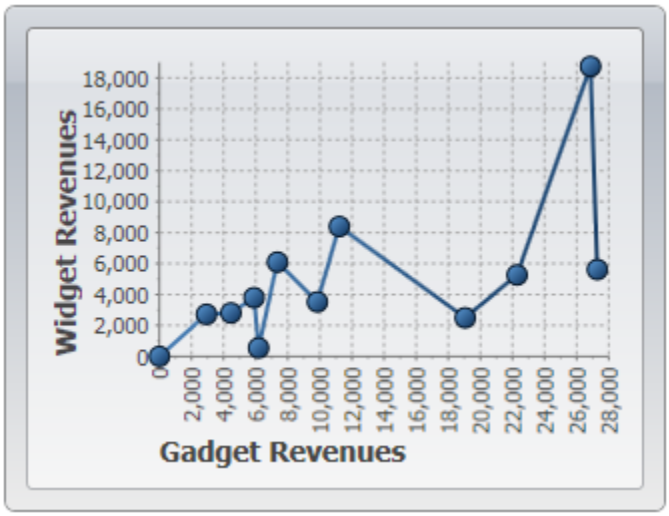
```
c1Chart.Theme = c1Chart.TryFindResource(new
ComponentResourceKey(typeof(C1.WPF.C1Chart.C1Chart),
"Office2007Black")) as ResourceDictionary;
        }
```

You can test it by running the program and changing the ChartType property to **XYPlot**, **LineSymbols**, or **Area** to create charts of different types. The result should be similar to the images below:

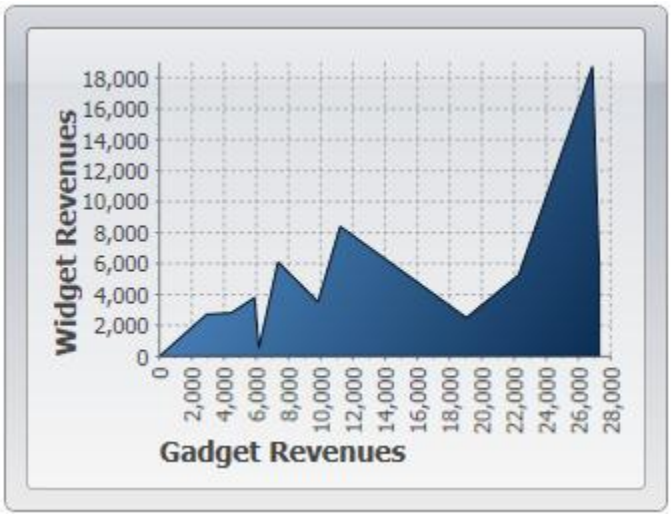
ChartType.XYPlot



ChartType.LineSymbols



ChartType.Area



The most appropriate chart type in this case is the first, an **XYPlot**. The chart shows a positive correlation between **Gadget** and **Widget** revenues.

This concludes the basic charting topic. You already have the tools you need to create all types of common charts.

Formatting Charts

The previous section introduced the Theme that you can use to select the appearance of your charts quickly and easily. The Theme and Palette properties offer a long list of built-in options that were carefully developed to provide great results with little effort from developers.

In most applications, you will choose the combination of settings for the **Theme** and **Palette** properties that is closest to the feel you want for your application, then customize a few items if necessary. Items you may want to customize include:

1. **Axis titles**: The axis titles are **UIElement** objects. You can customize them directly, and with complete flexibility. The chart samples used in the Common Usage for Basic 2D Charts (page 53) topic uses the **TextElement** objects, but you could use many other elements, including panels such as **Border** and **Grid** objects. For more information on axis titles, see Axis Title (page 101).

2. **Axis**: The chart samples used in the Common Usage for Basic 2D Charts (page 53) topic shows how you can customize axis scale, annotation angle, and annotation format. All these are accessible through the **Axis** object exposed by the AxisX and AxisY properties. For more information on C1Chart's axis, see Axes (page 98).

   The **C1Chart** control has the usual Font properties that determine how annotations are displayed along both axes (**FontFamily**, **FontSize**, etc). If you need more control over the appearance of the annotations, the Axis object also exposes an **AnnoTemplate** property that can be used to customize annotations even further.

3. **Grid lines**: Grid lines are controlled by the **Axis** properties. There are properties for the major and minor grid lines (MajorGridStrokeThickness, MajorGridStrokeThickness, MinorGridStrokeThickness, MinorGridStrokeThickness, and so on). For more information on grid lines, see Axis Grid Lines (page 104).
4. **Tick Marks**: Tick marks are also controlled by the **Axis** properties. There are properties for the major and minor ticks (MajorTickStroke, MajorTickThickness, MinorTickStroke, MinorTickThickness, and so on). For more information on tick marks, see Axis Tick Marks (page 102).

Chart Types

Using built-in types is the simplest way to set up the chart's appearance. For example, to set up a Stacked Bar chart, specify the corresponding string in the ChartType property:

```
<c1chart:C1Chart ChartType="BarStacked">
   ...
</c1chart:C1Chart>
```

The available chart types are specified by the members of enumeration ChartType.

The list of available built-in chart types is presented in the table below.

Name in gallery
Area
AreaSmoothed
AreaStacked
AreaStacked100pc
Bar
BarStacked
BarStacked100pc
Bubble
Candle
Column
ColumnStacked
ColumnStacked100pc
Gantt

HighLowOpenClose
Line
LineSmoothed
LineStacked
LineStacked100pc
LineSymbols
LineSymbolsSmoothed
LineSymbolsStacked
LineSymbolsStacked100pc
Pie
PieDoughnut
PieExploded
PieExplodedDoughnut
PolarLines
PolarLinesSymbols
PolarSymbols
Polygon
PolygonFilled
Radar
RadarFilled
RadarSymbols
Step
StepArea
StepSymbols
XYPlot
Area3D
Area3DSmoothed
Area3DStacked
Area3DStacked100pc
Bar3D
Bar3DStacked
Bar3DStacked100pc

Pie3D
Pie3DDoughnut
Pie3DExploded
Pie3DExplodedDougnut
Ribbon

Area Charts

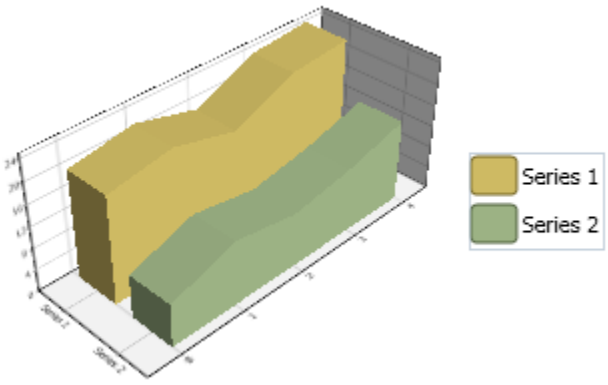
An **Area** chart draws each series as connected points of data, filled below the points. Each series is drawn on top of the preceding series. The series can be drawn independently or stacked. **Chart for WPF** supports the following types of **Area** charts:

- Area3D
- Area3Dsmoothed
- Area3Dstacked
- Area3Dstacked100pc
- AreaSmoothed
- AreaStacked
- AreaStacked100pc

3D Area Charts

Use the AreaShape3D class to access data points associated with the plot elements in the **3D Area** chart, get the value of the plot element when the mouse cursor is over it, get or set the size of plot elements in pixels, specify whether points are connected with smoothed lines.

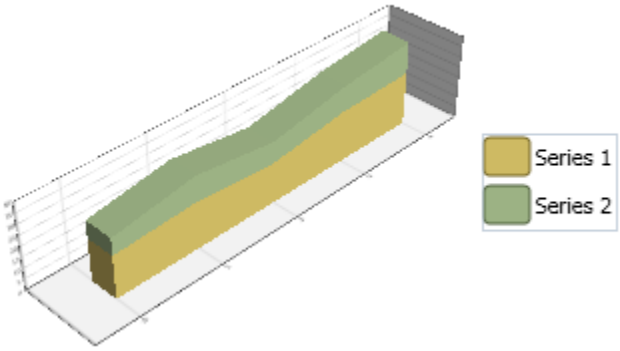
The following image represents the **3D Area** chart when you set the ChartType property to **Area3D**:



Area 3D Stacked Charts

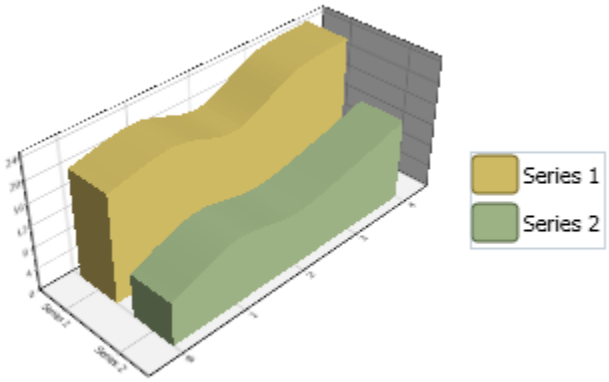
Use the **BaseRenderer** property and set the StackedOptions enumeration to create a specific stacking **Area** chart such as Stacked or Stacked 100%. Stacking charts represent the data by stacking the values for each series on top of the values from the previous series.

The following image represents the **Area 3D Stacked** chart when you set the ChartType property to **Area3Dstacked**:



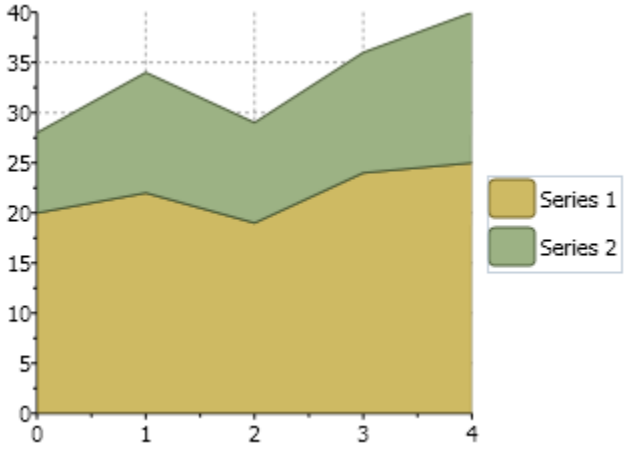
Area Smoothed

The following image represents the **Area Smoothed** chart when you set the ChartType property to **AreaSmoothed**:



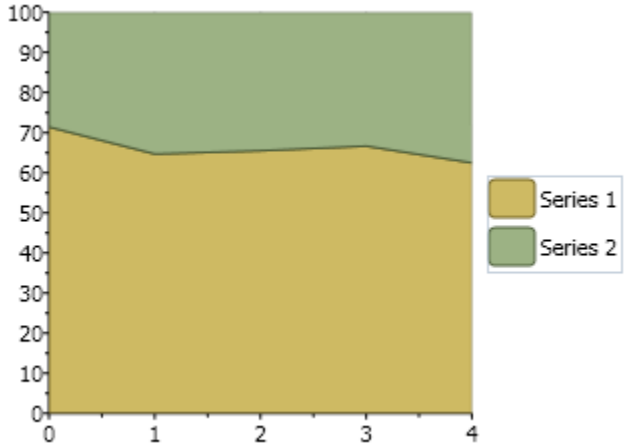
Area Stacked

The following image represents the **Area Stacked** chart when you set the ChartType property to **AreaStacked**:



Area Stacked 100 Percent

The following image represents the **Area Stacked 100 Percent** chart when you set the ChartType property to **AreaStacked100pc**:



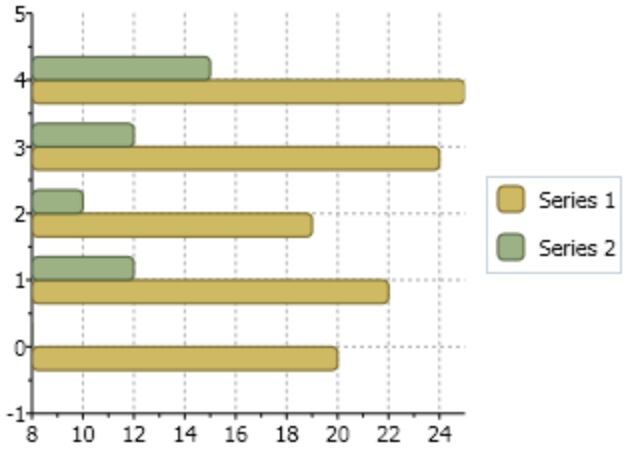
Bar Charts

Chart for WPF supports the following types of **Bar** charts:

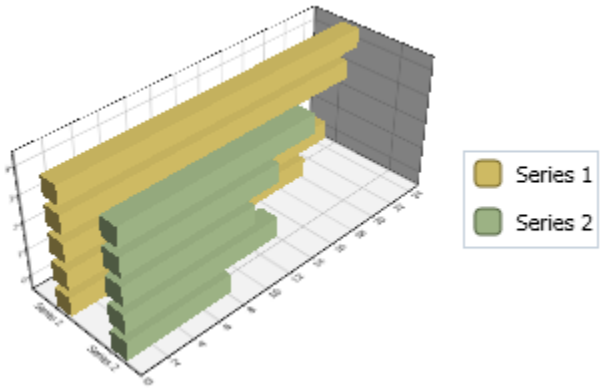
- Bar
- Bar3D
- Bar3Dstacked
- Bar3Dstacked100pc
- BarStacked
- BarStacked100pc

Bar

The following image represents the **Bar** chart when you set the ChartType property to **Bar**:

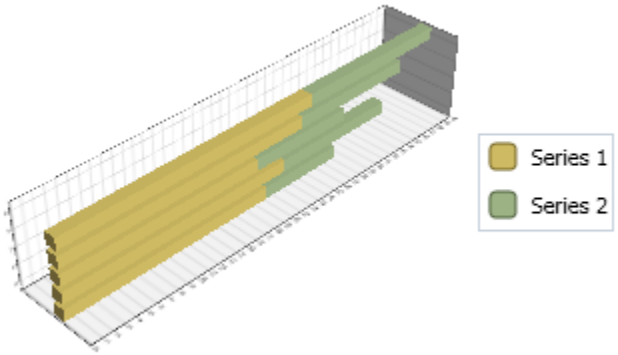


3D Bar The following image represents the **3D Bar** chart when you set the ChartType property to **Bar3D**:



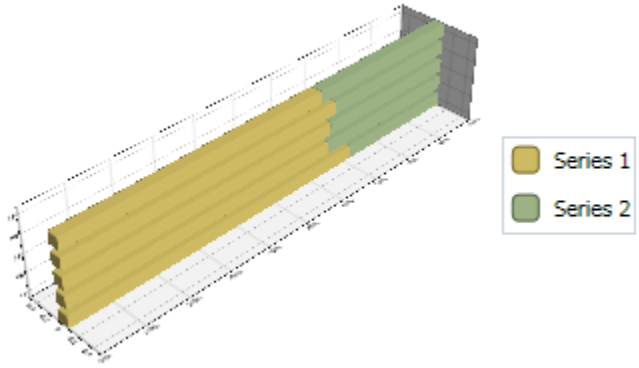
3D Bar Stacked

The following image represents the **3D Bar Stacked** chart when you set the ChartType property to **Bar3DStacked**:



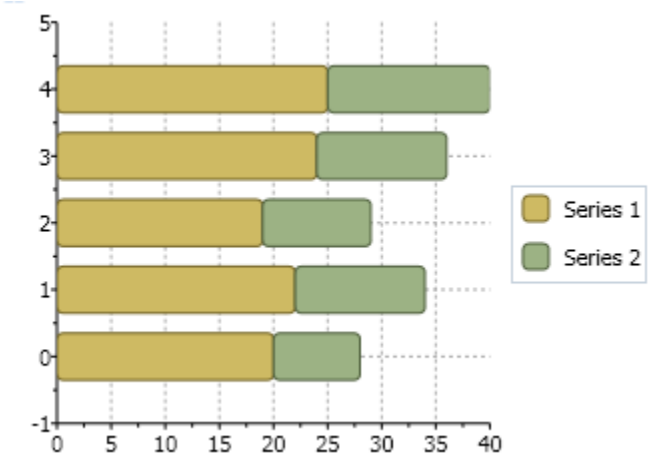
Bar 3D Stacked 100 Percent

The following image represents the **Bar 3D Stacked** 100% chart when you set the ChartType property to **Bar3Dstacked100pc**:



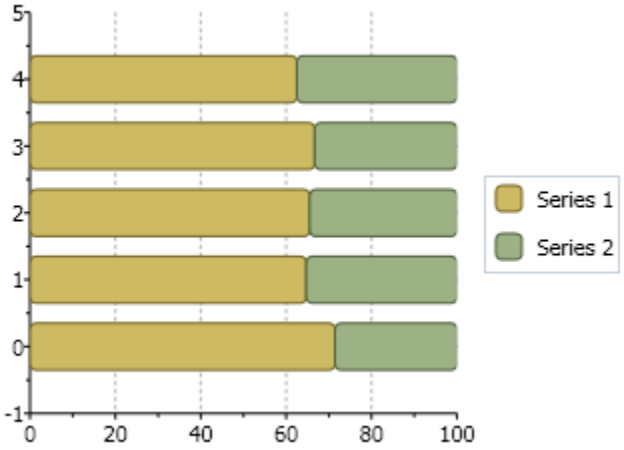
Bar Stacked

The following image represents the **Bar Stacked** chart when you set the ChartType property to **BarStacked**:



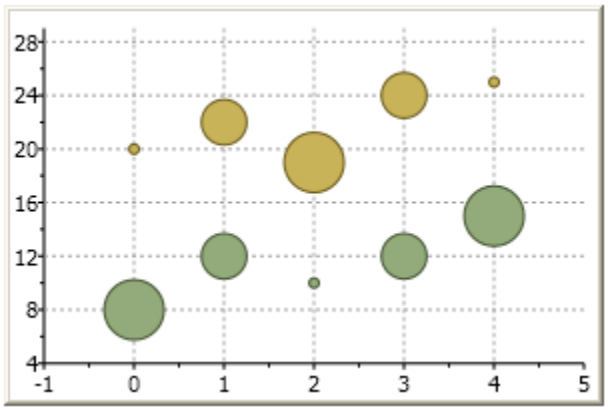
Bar Stacked 100 Percent

The following image represents the **Bar Stacked 100 Percent** chart when you set the ChartType property to **BarStacked100pc**:



Bubble Charts

The following image represents the **Bubble** chart when you set ChartType property to **Bubble**:



The following XAML code creates a **Bubble** chart:

```
  <c1chart:C1Chart ChartType="Bubble"
       c1chart:BubbleOptions.MinSize="5,5"
       c1chart:BubbleOptions.MaxSize="30,30"
       c1chart:BubbleOptions.Scale="Area">
    <c1chart:C1Chart.Data>
      <c1chart:ChartData>
        <c1chart:BubbleSeries Values="20 22 19 24 25" SizeValues="1 2 3
2 1" />
        <c1chart:BubbleSeries Values="8 12 10 12 15" SizeValues="3 2 1
2 3"/>
      </c1chart:ChartData>
    </c1chart:C1Chart.Data>
  </c1chart:C1Chart>
```

Financial Charts

C1Chart implements two types of financial chart: **Candle** and **HighLowOpenClose**. Both are commonly used to display variations in stock prices over a period of time.

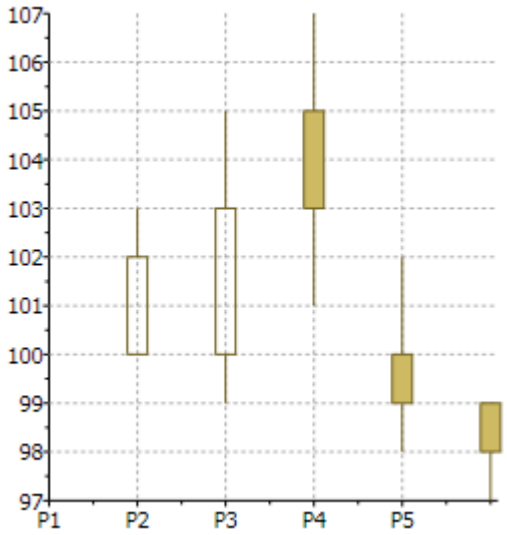
A **Candle** chart is a special type of **HiLoOpenClose** chart that is used to show the relationship between the open and close as well as the high and low. Like, **HiLoOpenClose** charts, **Candle** charts use the same price data (time, high, low, open, and close values) except they include a thick candle-like body that uses the color and size of the body to reveal additional information about the relationship between the open and close values. For example, long transparent candles show buying pressure and long filled candles show selling pressure.

The **Candle** chart is made up of the following elements: candle, wick, and tail. The candle or the body (the solid bar between the opening and closing values) represents the change in stock price from opening to closing. The thin lines, wick and tail, above and below the candle depict the high/low range. A hollow candle or transparent candle indicates a rising stock price (close was higher than open). In a hollow candle, the bottom of the body represents the opening price and the top of the body represents the closing price. A filled candle indicates a falling stock price (open was higher than close). In a filled candle the top of the body represents the opening price and the bottom of the body represents the closing price.

Candle Chart

The following image represents the **Candle** chart when you set ChartType property to **Candle** and specifiy the data values for the XValues, HighValues, LowValues, OpenValues, and CloseValues, like the following:

```
<c1chart:C1Chart ChartType="Candle">
    <c1chart:C1Chart.Data>
      <c1chart:ChartData>
        <c1chart:HighLowOpenCloseSeries
            XValues="1 2 3 4 5"
           HighValues="103 105 107 102 99"
           LowValues="100 99 101 98 97"
           OpenValues="100 100 105 100 99"
           CloseValues="102 103 103 99 98"
         />
      </c1chart:ChartData>
    </c1chart:C1Chart.Data>
  </c1chart:C1Chart>
```

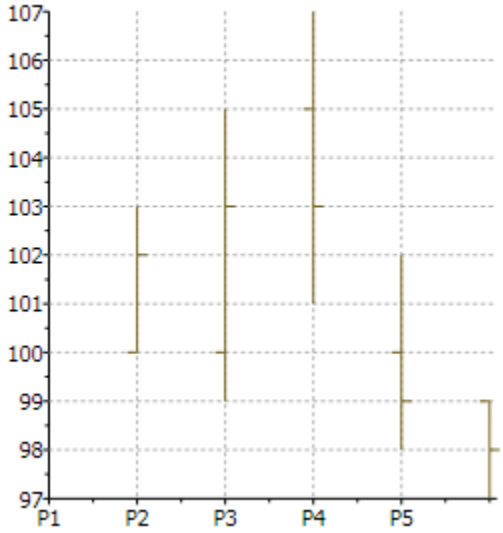


HighLowOpenClose Chart

The following image represents the **HighLowOpenClose** chart when you set ChartType property to **HighLowOpenClose** and specifiy the data values for the XValues, HighValues, LowValues, OpenValues, and CloseValues, like the following:

```
<c1chart:C1Chart ChartType="HighLowOpenClose">
    <c1chart:C1Chart.Data>
      <c1chart:ChartData>
        <c1chart:HighLowOpenCloseSeries
            XValues="1 2 3 4 5"
           HighValues="103 105 107 102 99"
           LowValues="100 99 101 98 97"
           OpenValues="100 100 105 100 99"
           CloseValues="102 103 103 99 98"
         />
      </c1chart:ChartData>
    </c1chart:C1Chart.Data>
```

```
  </c1chart:C1Chart>
```



The difference between common chart types and financial charts is that **Candle** and **HighLowOpenClose** charts require a special type of data series object, the **HighLowOpenCloseSeries**. In this type of data series, each point corresponds to a period (typically one day) and contains five values:

- Time
- Price at the beginning of period (Open)
- Price at the end of period (Close)
- Minimum price during period (Low)
- Maximum price during period (High)

To create financial charts you need to provide all these values.

For example, if the values were provided by the application as collections, then you could use the code below to create the data series:

```
// create data series
HighLowOpenCloseSeries ds = new HighLowOpenCloseSeries();
ds.XValuesSource = dates; // dates are along x-axis
ds.OpenValuesSource = open;
ds.CloseValuesSource = close;
ds.HighValuesSource = hi;
ds.LowValuesSource = lo;

// add series to chart
chart.Data.Children.Add(ds);

// set chart type
chart.ChartType = isCandle
  ? ChartType.Candle
  : ChartType.HighLowOpenClose;
```

Another option is to use data-binding. For example, if the data is available as a collection of **StockQuote** objects such as:

```
public class Quote
{
  public DateTime Date { get; set; }
  public double Open { get; set; }
  public double Close { get; set; }
  public double High { get; set; }
  public double Low { get; set; }
}
```

Then the code that creates the data series would be as follows:

```
// create data series
HighLowOpenCloseSeries ds = new HighLowOpenCloseSeries();

// bind all five values
ds.XValueBinding = new Binding("Date"); // dates are along x-axis
ds.OpenValueBinding = new Binding("Open");
ds.CloseValueBinding = new Binding("Close");
ds.HighValueBinding = new Binding("High");
ds.LowValueBinding = new Binding("Low");

// add series to chart
chart.Data.Children.Add(ds);

// set chart type
chart.ChartType = isCandle
  ? ChartType.Candle
  : ChartType.HighLowOpenClose;
```

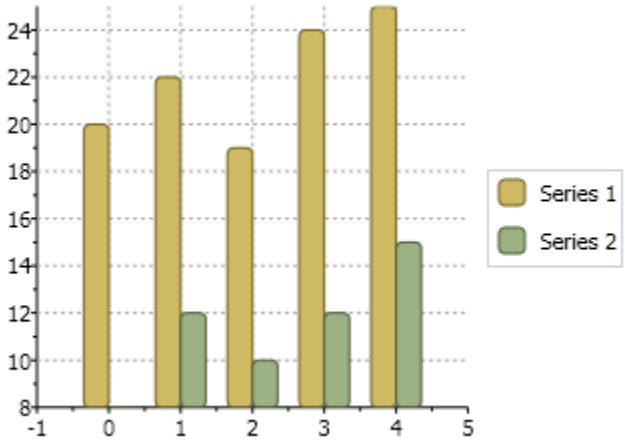
Column Charts

Chart for WPF supports the following types of **Column** charts:

- Column
- Column3D
- Column3DStacked
- Column3Dstacked100pc
- ColumnStacked
- ColumnStacked100pc

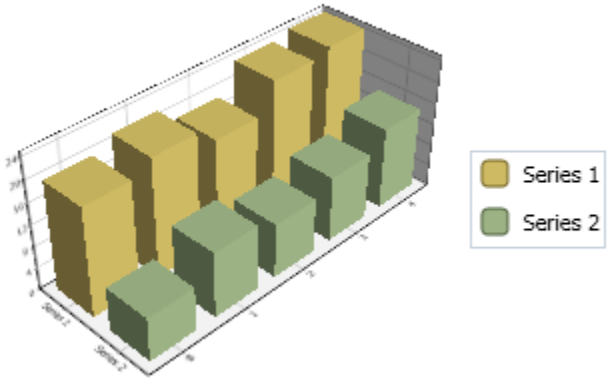
Column

The following image represents the **Column** chart when you set the ChartType property to **Column**:



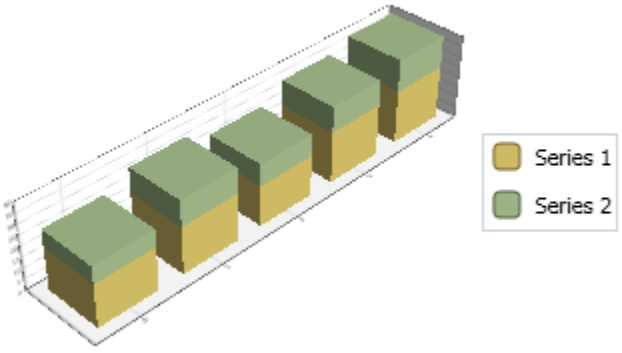
3D Column

The following image represents the **3D Column** chart when you set the ChartType property to **Column3D**:



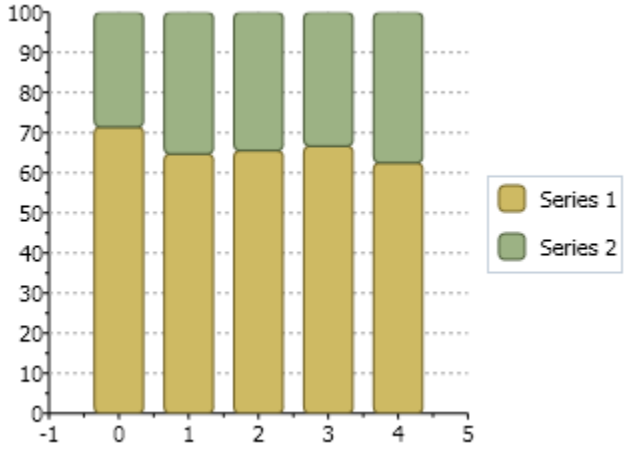
3D Column Stacked

The following image represents the **3D Column Stacked** chart when you set the ChartType property to **Column3DStacked**:



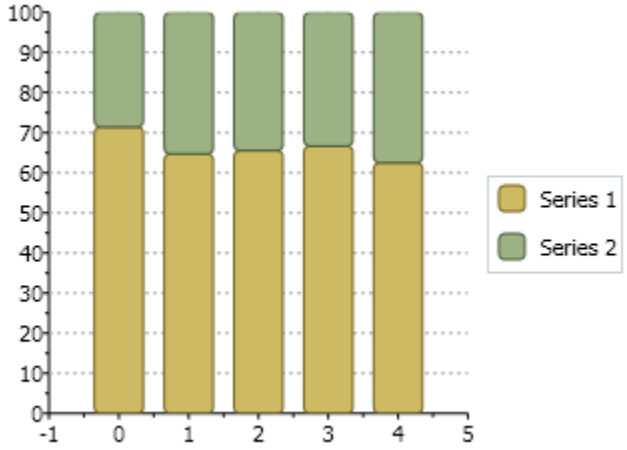
3D Column Stacked 100%

The following image represents the **3D Column Stacked 100%** chart when you set the ChartType property to **Column3Dstacked100pc**:



Column Stacked 100%

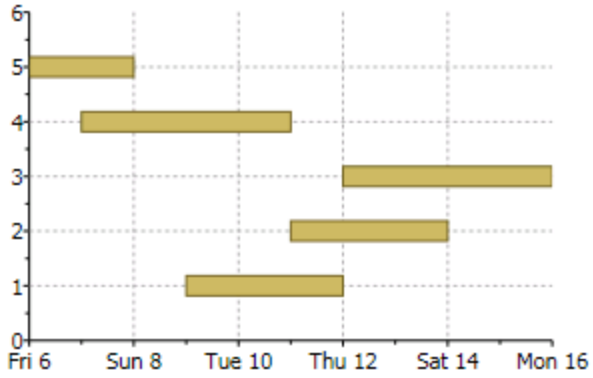
The following image represents the **Column Stacked 100%** when you set the ChartType property to **ColumnStacked100pc**:



Gantt Charts

Gantt charts use data series objects of type **HighLowSeries**. Each data series represents a single task, and each task has a set of start and end values. Simple tasks have one start value and one end value. Tasks that are composed of multiple sequential sub-tasks have multiple pairs of start and end values.

The following image represents a **Gantt** chart:



To demonstrate **Gantt** charts, let us start by defining a **Task** object:

```
class Task
{
  public string Name { get; set; }
  public DateTime Start { get; set; }
  public DateTime End { get; set; }
  public bool IsGroup { get; set; }
  public Task(string name, DateTime start, DateTime end, bool isGroup)
  {
    Name = name;
    Start = start;
    End = end;
    IsGroup = isGroup;
  }
}
```

Next, let us define a method that creates a set of **Task** objects that will be shown as a **Gantt** chart:

```
Task[] GetTasks()
{
  return new Task[]
  {
    new Task("Alpha", new DateTime(2008,1,1), new DateTime(2008,2,15),
true),
    new Task("Spec", new DateTime(2008,1,1), new DateTime(2008,1,15),
false),
    new Task("Prototype", new DateTime(2008,1,15), new
DateTime(2008,1,31), false),
    new Task("Document", new DateTime(2008,2,1), new
DateTime(2008,2,10), false),
    new Task("Test", new DateTime(2008,2,1), new DateTime(2008,2,12),
false),
    new Task("Setup", new DateTime(2008,2,12), new DateTime(2008,2,15),
false),

    new Task("Beta", new DateTime(2008,2,15), new DateTime(2008,3,15),
true),
    new Task("WebPage", new DateTime(2008,2,15), new
DateTime(2008,2,28), false),
    new Task("Save bugs", new DateTime(2008,2,28), new
DateTime(2008,3,10), false),
```

```
    new Task("Fix bugs", new DateTime(2008,3,1), new
DateTime(2008,3,15), false),
    new Task("Ship", new DateTime(2008,3,14), new DateTime(2008,3,15),
false),
  };
}
```

Now that the tasks have been created, we are ready to create the **Gantt** chart:

```
private void CreateGanttChart()
{
  // clear current chart
  c1Chart.Reset(true);

// set chart type
  c1Chart.ChartType = ChartType.Gantt;

// populate chart
  var tasks = GetTasks();
  foreach (var task in tasks)
  {
  // create one series per task
    var ds = new HighLowSeries();
    ds.Label = task.Name;
    ds.LowValuesSource = new DateTime[] { task.Start };
    ds.HighValuesSource = new DateTime[] { task.End };
    ds.SymbolSize = new Size(0, task.IsGroup ? 30 : 10);

  // add series to chart
    c1Chart.Data.Children.Add(ds);
  }

// show task names along Y axis
  c1Chart.Data.ItemNames =
    (from task in tasks select task.Name).ToArray();

// customize Y axis
  var ax = c1Chart.View.AxisY;
  ax.Reversed = true;
  ax.MajorGridStroke = null;

// customize X axis
  ax = c1Chart.View.AxisX;
  ax.MajorGridStrokeDashes = null;
  ax.MajorGridFill = new SolidColorBrush(Color.FromArgb(20, 120, 120,
120));
  ax.Min = new DateTime(2008, 1, 1).ToOADate();
}
```

After clearing the **C1Chart** and setting the chart type, the code enumerates the tasks and creates one **HighLowSeries** for each. In addition to setting the series **Label**, **LowValuesSource** and **HighValuesSource** properties, the code uses the SymbolSize property to set the height of each bar. In this sample, we define some tasks as "Group" tasks, and make them taller than regular tasks.

Next, we use a Linq statement to extract the task names and assign them to the ItemNames property. This causes **C1Chart** to display the task names along the Y axis.

Finally, the code customizes the axes. The Y axis is reversed so the first task appears at the top of the chart. The axes are configured to show vertical grid lines and alternating bands.

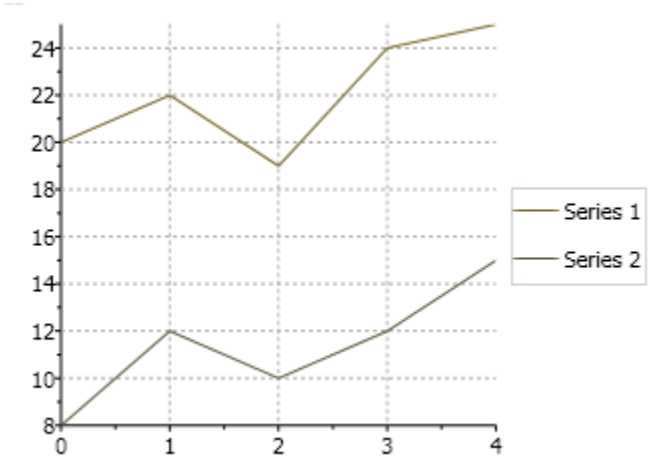
Line Charts

Chart for WPF supports the following types of **Line** charts:

- Line
- LineSmoothed
- LineStacked
- LineStacked100pc
- LineSymbols
- LineSymbolsSmoothed
- LineSymbolsStacked
- LineSymbolsStacked100pc

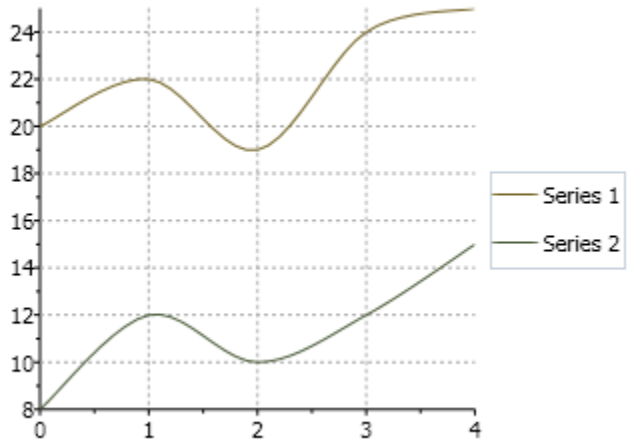
Line

The following image represents the **Line** chart when you set the ChartType property to **Line**:



Line Smoothed

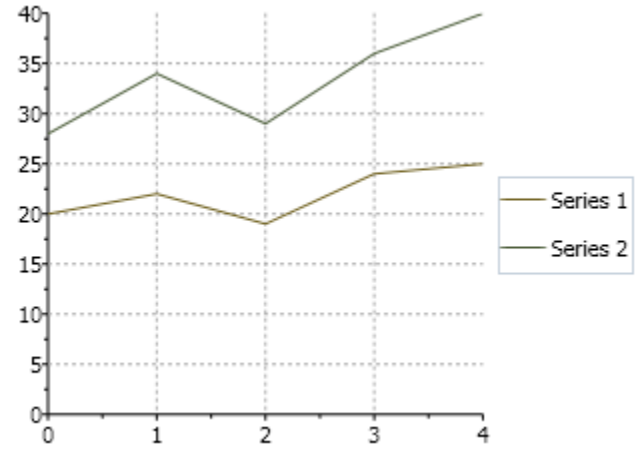
The following image represents the **Line Smoothed** chart when you set the ChartType property to **LineSmoothed**:



Line Stacked

Select the **LineStacked** member from the ChartType enumeration to create a specific stacking Line chart. Stacking charts represent the data by stacking the values for each series on top of the values from the previous series.

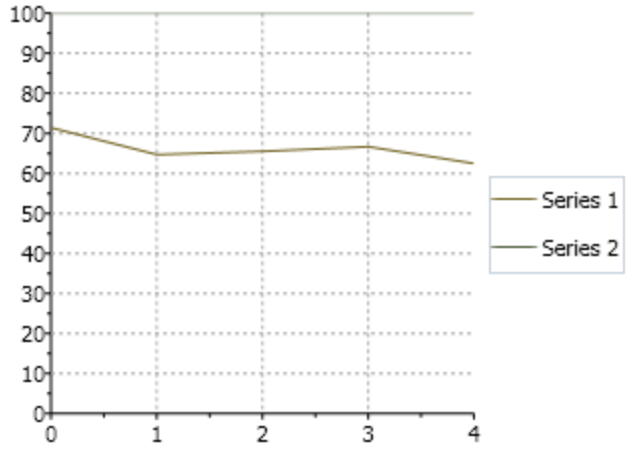
The following image represents the **Line Stacked** chart when you set the ChartType property to **LineStacked**:



Line Stacked 100%

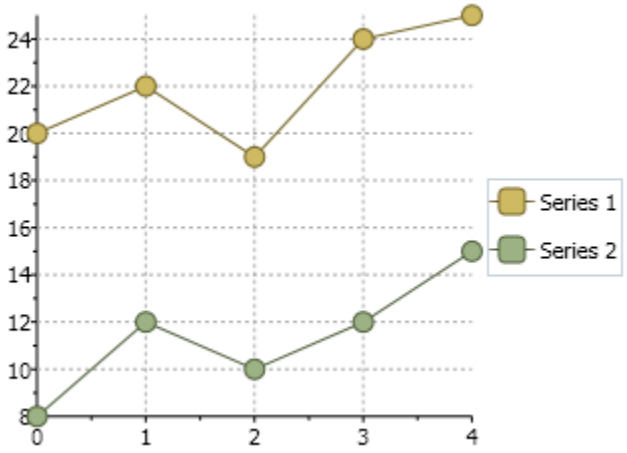
Select the **LineStacked100pc** member from the ChartType enumeration to create a specific stacking Line chart. Stacking charts represent the data by stacking the values for each series on top of the values from the previous series.

The following image represents the Line Stacked 100% chart when you set the ChartType property to **LineStacked100pc**:



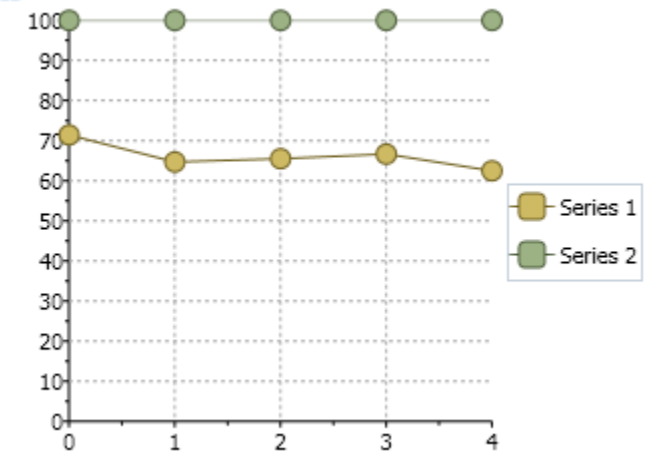
Line Symbols

The following image represents the **Line Symbols** when you set the ChartType property to **LineSymbols**:



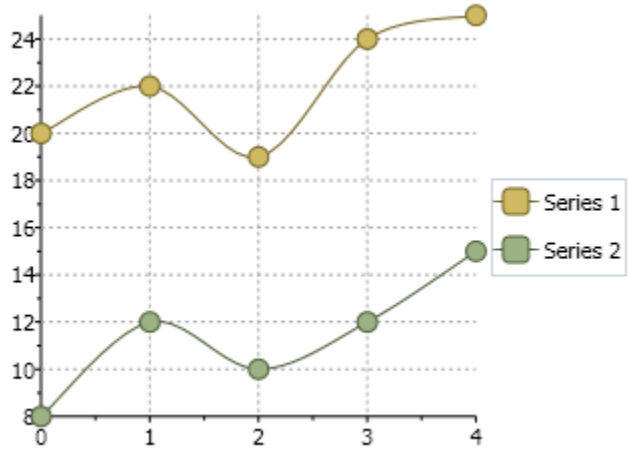
Line Symbols Stacked

The following image represents the **Line Symbols Stacked** chart when you set the ChartType property to **LineSymbolsStacked**:



Line Symbols Smoothed

The following image represents the **Line Symbols Smoothed** when you set the ChartType property to **LineSymbolsSmoothed**:



Pie Charts

Pie charts are commonly used to display simple values. They are visually appealing and often displayed with 3D effects such as shading and rotation.

Pie charts have one significant difference when compared to other **C1Chart** chart types in Pie charts; each series represents one slice of the pie. Therefore, you will never have Pie charts with a single series (they would be just circles). In most cases, Pie charts have multiple series (one per slice) with a single data point in each series. **C1Chart** represents series with multiple data points as multiple pies within the chart.

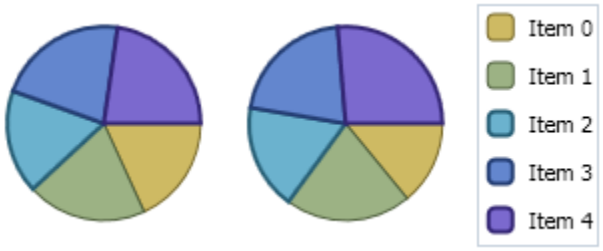
Chart for WPF supports the following types of Pie charts:

- Pie
- 3D Pie
- 3D Doughnut Pie
- 3D Exploded Pie
- 3D Exploded Doughnut Pie

- Dougnut Pie
- Exploded Pie
- Exploded Doughnut Pie

Pie

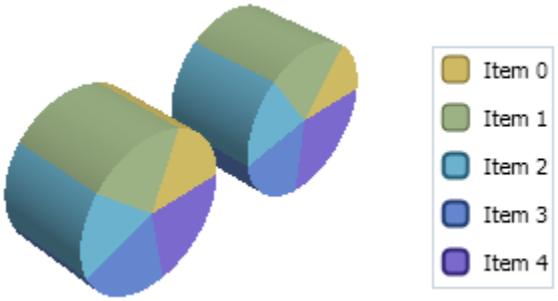
The following image represents the **Pie** chart when you set the ChartType property to **Pie**:



3D Pie

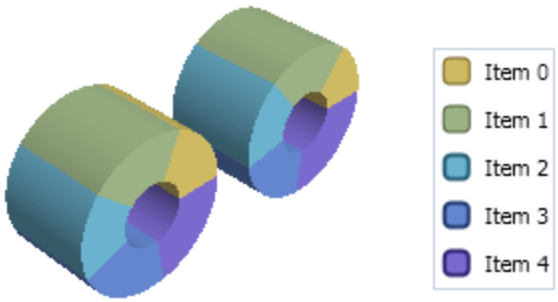
Use the Pie3D class to access data points associated with the plot elements in the 3D Pie chart, get the value of the plot element when the mouse cursor is over it, get or set the size of plot elements in pixels, and specify whether points are connected with smoothed lines.

The following image represents the **3D Pie** chart when you set the ChartType property to **Pie3D**:



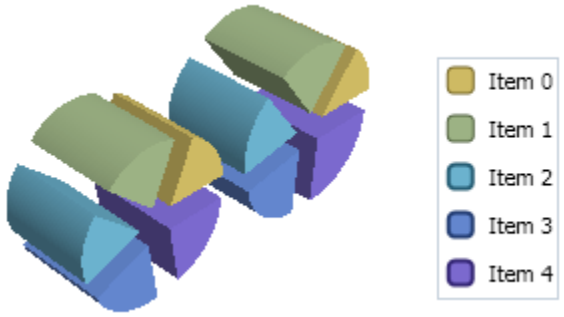
3D Doughnut Pie

The following image represents the **3D Doughnut Pie** chart when you set the ChartType property to **Pie3DDoughnut**:



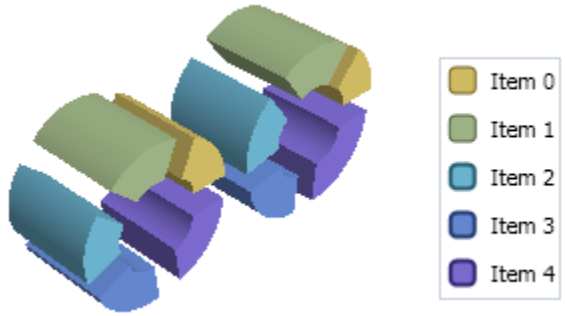
3D Exploded Pie

The following image represents the **3D Exploded Pie** chart when you set the ChartType property to **Pie3DExploded**:



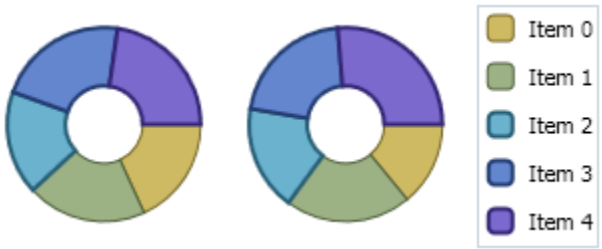
3D Exploded Doughnut Pie

The following image represents the **3D Exploded Doughnut Pie** chart when you set the ChartType property to **Pie3DExplodedDoughnut**.



Doughnut Pie

The following image represents the **Doughnut Pie** chart when you set the ChartType property to **PieDoughnut**.



Exploded Pie

The following image represents the **Exploded Pie** chart when you set ChartType property to **PieExploded**:

Exploded Doughnut Pie

The following image represents the **Exploded Doughnut Pie** chart when you set ChartType property to **PieExplodedDoughnut**.

Special Pie Chart Properties

Pie charts are quite different from the other chart types since they do not follow the concept of a two-dimensional grid or axes. Altering the diameter of the pie or the properties of the exploding slices can be accomplished with the properties of the Pie class.

Starting Angle

Use the **PieOptions.StartingAngleAttached** property to specify the angle at which the slices for the first series start. The default angle is 0 degrees. The angle represents the arc between the most clockwise edge of the first slice and the right horizontal radius of the pie, as measured in the counter-clockwise direction.

Exploding Pies

A slice of a Pie chart can be emphasized by exploding it, which extrudes the slice from the rest of the pie. Use the Offset property of the series to set the exploded slice's offset from the center of the pie. The offset is measured as a percentage of the radius of the pie.

Polar and Radar Charts

Polar Charts

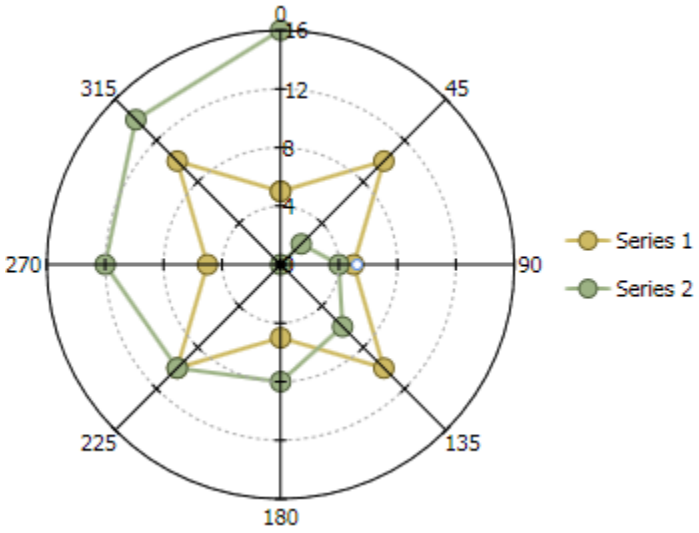
A **Polar** chart draws the x and y coordinates in each series as (theta,r), where theta is amount of rotation from the origin and r is the distance from the origin. Theta may be specified in either degrees (default) or radians. Since the X-axis is a circle, the X-axis maximum and minimum values are fixed.

Polar charts can not be combined with any other chart type in the same chart area.

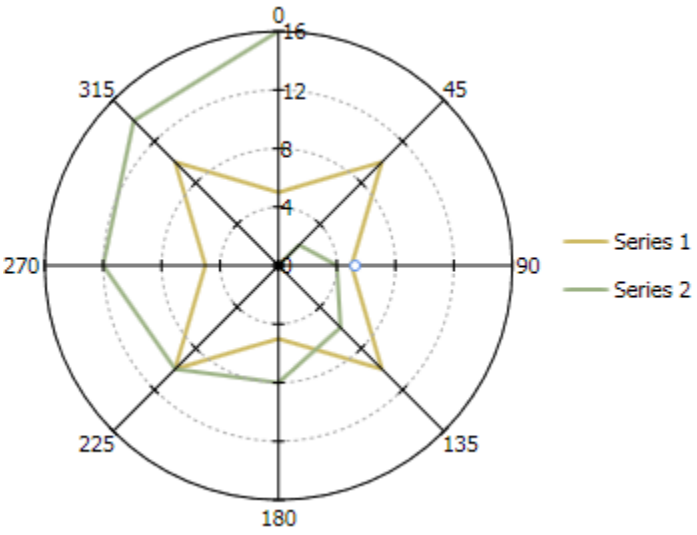
The following image represents the **Polar** chart when you set ChartType property to **PolarLines**.

The following images represent the different types of **Polar** charts when you set ChartType property to **PolarLinesSymbol**, **PolarLines**, **PolarSymbols** and specifiy the data values for the **XYDataSeries**, like the following:

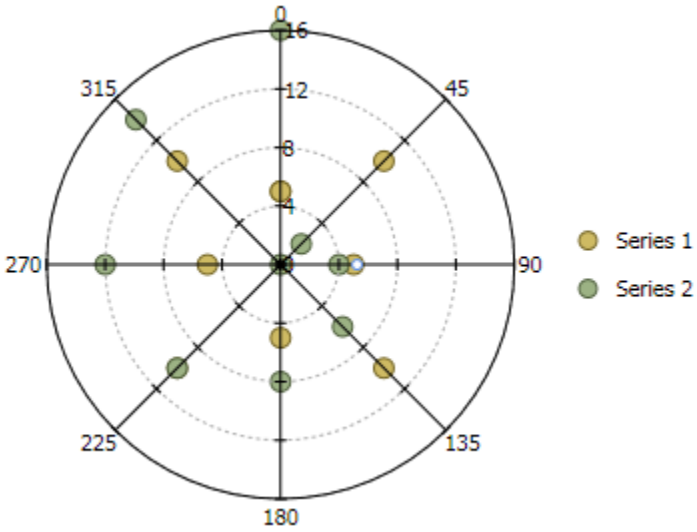
```
<c1chart:C1Chart Name="c1Chart1" ChartType="PolarLinesSymbols">
  <c1chart:C1Chart.Data>
   <c1chart:ChartData>
    <c1chart:XYDataSeries Label="Series 1" Values="5 10 5 10 5 10 5 10
5"
                                XValues="0 45 90 135 180 225 270 315
0"/>
    <c1chart:XYDataSeries Label="Series 2" Values="0 2 4 6 8 10 12 14
16"
                               XValues="0 45 90 135 180 225 270 315
0"/>
   </c1chart:ChartData>
  </c1chart:C1Chart.Data>
  <c1chart:C1ChartLegend DockPanel.Dock="Right" /> </c1chart:C1Chart>
```



The following image represents the **Polar** Lines chart with symbols and lines when you set ChartType property to **PolarLines**.



The following image represents the **Polar Symbols** chart when you set ChartType property to **PolarSymbols**.

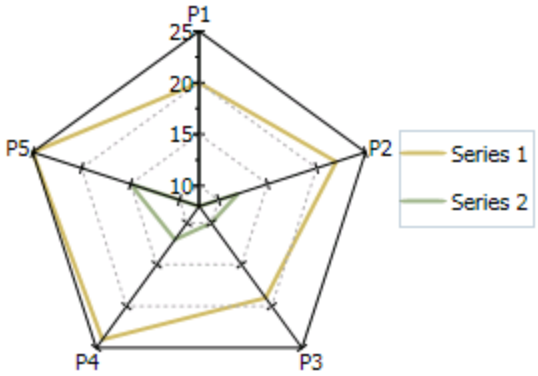


Radar Charts

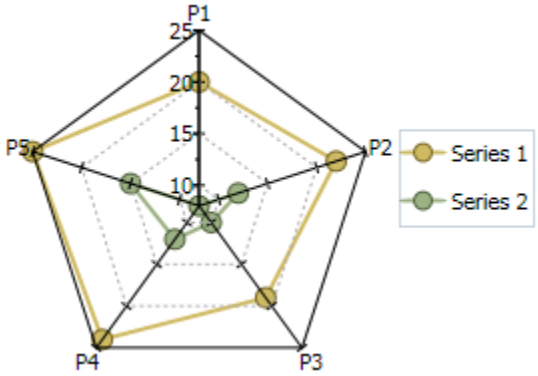
A **Radar** chart is a variation of a **Polar** chart. A **Radar** chart draws the y value in each data set along a radar line. If the data has n unique points, then the chart plane is divided into n equal angle segments, and a radar line is drawn (representing each point) at n/360 degree increments. By default, the radar line representing the first point is drawn vertically (at 90 degrees).

The labels for radar chart can be set using ItemNames property. These labels are located at end of each radial line.

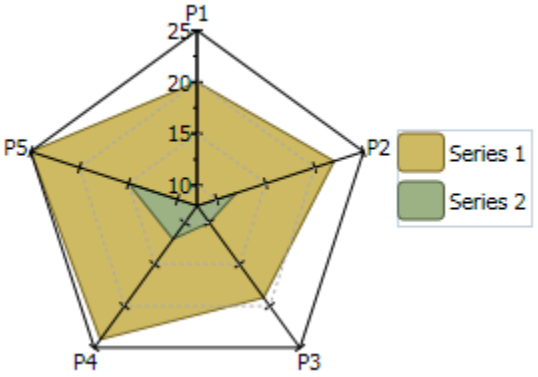
The following image represents the Radar chart when you set ChartType property to **Radar**.



The following image represents the **Radar** chart with symbols when you set ChartType property to **RadarSymbols**.



The following image represents the filled **Radar** chart when you set ChartType property to **RadarFilled**.



Special Polar and Radar Chart Properties

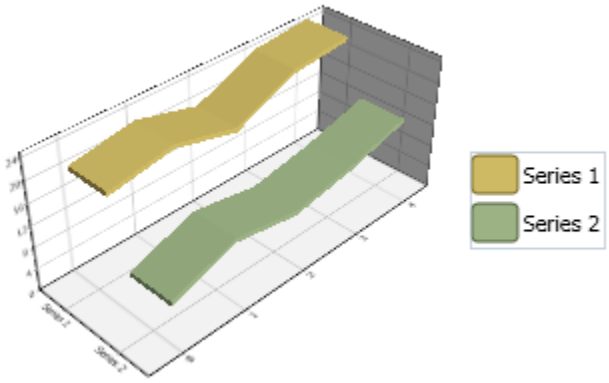
The **Polar** and **Radar** charts have special properties to chart the degrees of the **Radar** and set the starting angle.

Setting the Starting Angle

The **PolarRadarOptions.StartingAngle Attached** property of the PolarRadarOptions class sets the starting angle for Polar and Radar charts. The default setting for this property is 0. Setting this property to a value other than 0 will move the origin of the chart counter-clockwise by the specified degrees. For instance, setting the **PolarRadarOptions.StartingAngle Attached** property to 90, the Polar or Radar chart rotates 90 degrees in the counter-clockwise direction.

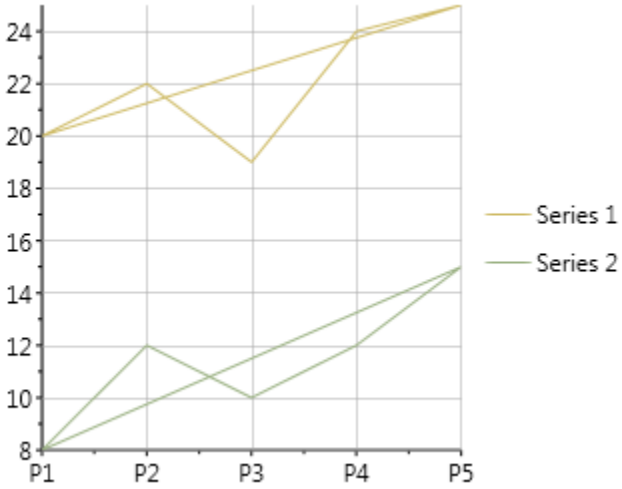
3D Ribbon Chart

The following image represents the **3D Ribbon** chart when you set ChartType property to **Ribbon**:

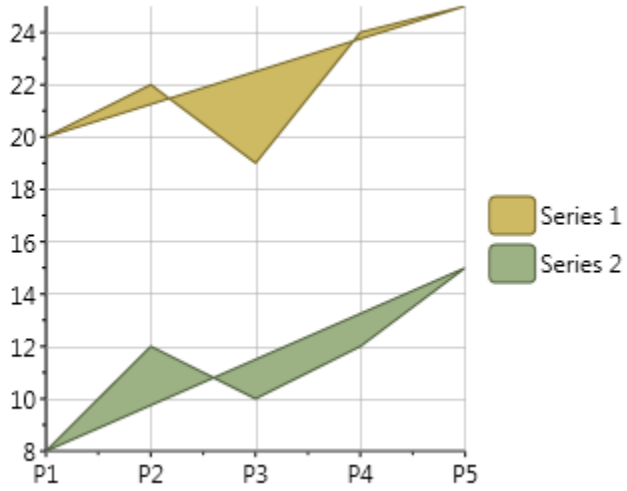


Polygon Chart

The following image represents the **Polygon** chart when you set ChartType property to **Polygon**:



The following image represents the **Polygon Filled** chart when you set ChartType property to **PolygonFilled**:



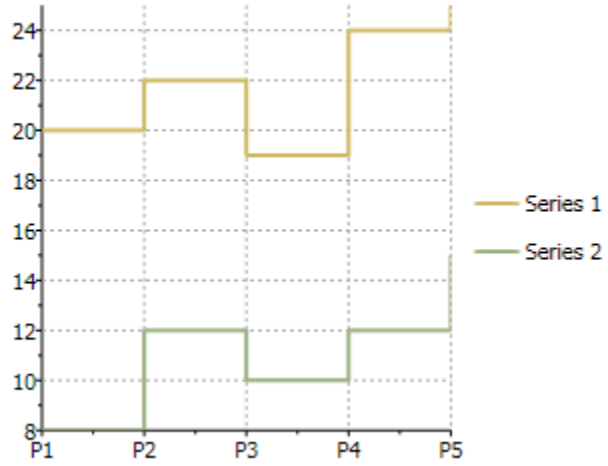
Step Chart

A **Step** chart is a form of a **XY Plot** chart. **Step** charts are often used when Y values change by discreet amounts, at specific values of X with a sudden change of value. A simple, everyday example would be a plot of a checkbook balance with time. As each deposit is made, and each check is written, the balance (Y value) of the check register changes suddenly, rather than gradually, as time passes (X value). During the time that no deposits are made, or checks written, the balance (Y value) remains constant as time passes.

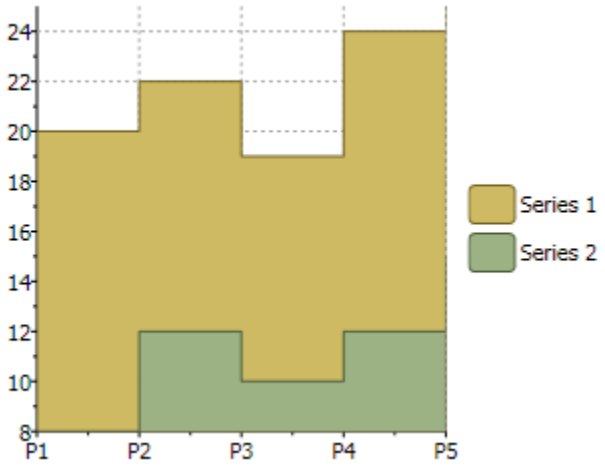
Similar to Line and XY plots, the appearance of the step chart can be customized by using the Connection and Symbol properties for each series by changing colors, symbol size, and line thickness. Symbols can be removed entirely to emphasize the relationship between points or included to indicate actual data values. If data holes are present, the step chart behaves as expected, with series lines demonstrating known information up to the X value of the data hole. Symbols and lines resume once non-hole data is again encountered.

As with most **XY** style plots, step charts can be stacked when appropriate.

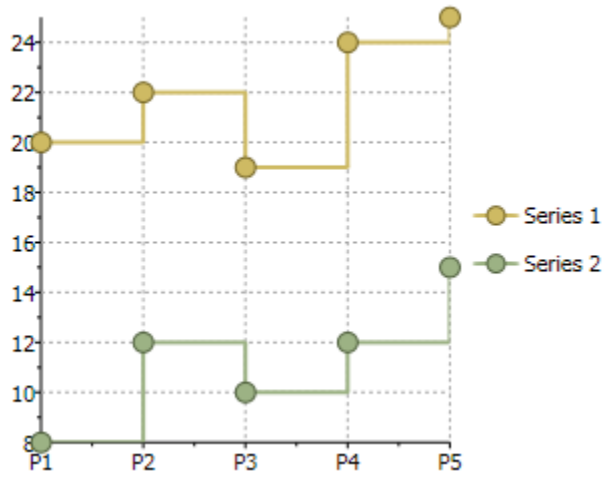
The following image represents the **Step** chart when you set ChartType property to **Step**.



The following image represents the **Area Step** chart with symbols when you set ChartType property to **StepArea**.



The following image represents the **Symbols Step** chart when you set ChartType property to **StepSymbols**.



XYPlot Chart

The **XYPlot** is also known as a **Scatter** plot chart. For more information on the XYPlot chart see XY Charts (page 62).

The following image represents the **XYPlot** chart when you set ChartType property to **XYPlot**:

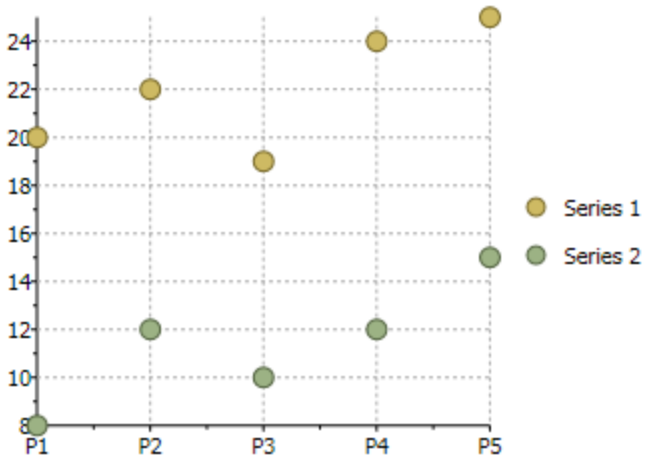


Chart Data Series

One of the more important objects in **C1Chart** is the data series. The data series contains all of the data to be included in the chart and many important data-related properties.

Chart Data Series Types

C1Chart provides the following dataseries classes to effectively handle different data types:

- BubbleSeries
- DataSeries
- HighLowOpenCloseSeries
- HighLowSeries
- XYDataSeries
- XYZDataSeries

The **Label** property in the DataSeries class represents the label for the Series name in the Chart Legend.

There are several **DataSeries** classes inherited from the same base class DataSeries, each of them combines several data sets depending on appropriate data nature. For instance, the XYDataSeries provides two sets of data values that correspond to x- and y-coordinates. The list of available data series classes is presented in the table below.

Class	Values properties	Value binding property
DataSeries	Values, ValuesSource	ValueBinding
XYDataSeries	Values, ValuesSource XValues, XValuesSource	ValueBinding XValueBinding
HighLowSeries	Values, ValuesSource XValues, XValuesSource HighValues, HighLowSeries.HighValuesSource LowValues, HighLowSeries.LowValuesSource	ValueBinding XValueBinding HighValueBinding LowValueBinding

HighLowOpenCloseSeries	Values, ValuesSource XValues, XValuesSource HighValues, HighLowSeries.HighValuesSource LowValues, HighLowSeries.LowValuesSource OpenValues, HighLowOpenCloseSeries.OpenValuesSource CloseValues, HighLowOpenCloseSeries.CloseValuesSource	ValueBinding XValueBinding HighValueBinding LowValueBinding OpenValueBinding CloseValueBinding

Each data series class may have its own default template for plotting, for instance HighLowOpenCloseSeries displays financial data as a set of lines that mark high, low, open and close values.

Chart Data Series Appearance

The appearance of each data series is determined by three groups of properties in the DataSeries class: **Symbol**, **Connection**, and **ConnectionArea**. These properties affect different parts of the chart depending on the chart type.

The **Symbol** properties determine the shape, size, outline, and fill properties of the symbols drawn at each data point. They apply to chart types that display symbols, including **Line**, **Area**, and **XYPlot** charts. The **Symbol** properties also control the appearance of bars in **Bar** and **Column** charts.

The **Connection** properties determine the outline and fill properties of the lines drawn between pairs of data points. They apply to all collection of points for data series. For line charts the connection is the line which connects points, for area charts the connection is the area including the outline below the data points.

Differences Between DataSeries and XYDataSeries

DataSeries has only one logical set of data values - Values(y-values).

In this case x-values are generated automatically(0,1,2...), also you can specify text labels for x-axis using **Data.ItemNames** property.

XYDataSeries has two sets of data values - Values(y-values) and XValues.

Render Mode Limitations for Data Series

The fast render mode has several limitations:

- Labels, tooltips, and the **PlotElement.Loaded** event are not supported.
- It's implemented only for line and symbol(XYPlot) charts(and their combination). These chart types are commonly used in case of large data.

Chart Panel

The ChartPanel is a container for the UI elements (ChartPanelObject's) that can be positioned using data coordinates. The ChartPanel object includes two unique properties: Chart and Children. The Chart gets or sets the parent chart and the Children property gets the collection of child elements.

To use chart panel with chart it's necessary to add the panel to the Layers collection of ChartView:

```
<c1chart:C1Chart x:Name="chart">
   <c1chart:C1Chart.View>
     <c1chart:ChartView>
```

```
      <c1chart:ChartView.Layers>
        <c1chart:ChartPanel >

         <!-- ChartPanelObjects -->

        </c1chart:ChartPanel>
      </c1chart:ChartView.Layers>
    </c1chart:ChartView>
  </c1chart:C1Chart.View>
</c1chart:C1Chart>
```

The ChartPanelObject class defines the element of the chart panel. The ChartPanelObject includes three unique properties: Action, Attach, and DataPoint properties.

Using **HorizontalAlignment** / **VerticalAlignment** properties it's possible to adjust relative position to the element and the related data point property, DataPoint. The **Content** property of the ChartPanelObject can be set to any UIElement.

The following XAML defines text label with its left-bottom corner at x=0, y=0 in data coordinates:

```
  <c1chart:ChartPanelObject DataPoint="0,0"
VerticalAlignment="Bottom">
      <TextBlock Text="Zero"/>
  </c1chart:ChartPanelObject>
```

Note: It is not necessary to specify both coordinates. If the coordinate is set to double.NaN then the element does not have specific x- or y- coordinates.

We can create horizontal marker at y=0. Note that the **HorizontalAlignment** property is set to **Stretch** and the element fills the width of the plot area.

```
  <!-- horizontal line -->
  <c1chart:ChartPanelObject DataPoint="NaN,0"
HorizontalAlignment="Stretch">
    <Border BorderBrush="Red" BorderThickness="0,2,0,0"
Margin="0,-1,0,0" />
  </c1chart:ChartPanelObject>
```

The following sample here creates a vertical marker:

```
  <!-- vertical line -->
  <c1chart:ChartPanelObject DataPoint="0,NaN"
VerticalAlignment="Stretch">
    <Border BorderBrush="Red" BorderThickness="2,0,0,0"
Margin="-1,0,0,0" />
  </c1chart:ChartPanelObject>
```

Note: The chart panel objects support only the main axes. For auxiliary axes you need to perform coordinate conversion.

Mouse Interaction with ChartPanel

The ChartPanel panel has support of mouse interaction. The ChartPanelAction enumeration specifies possible action for the chart panel objects. The ChartPanelAction enumeration includes the following members:

Member name	Description
None	No action.
MouseMove	Follow mouse.

LeftMouseButtonDrag	Can be dragged with left mouse button.
RightMouseButtonDrag	Can be dragged with right mouse button.

Using the Action property we can make a draggable element or element that follows the mouse pointer. For examplle, adding Action to the previous sample we get the marker that can be moved by user.

```
   <!-- vertical line -->
   <c1chart:ChartPanelObject DataPoint="0,NaN"
VerticalAlignment="Stretch"
   Action="LeftMouseButtonDrag" >
     <Border BorderBrush="Red" BorderThickness="3,0,0,0"
Margin="-1.5,0,0,0" />
   </c1chart:ChartPanelObject>
```

Using data binding it's easy to add label that shows the current coordinate:

```
   <!-- vertical line with coordinate label -->
   <c1chart:ChartPanelObject x:Name="xmarker" DataPoint="0,NaN"
VerticalAlignment="Stretch"
       Action="LeftMouseButtonDrag">
     <Border BorderBrush="Red" BorderThickness="3,0,0,0"
Margin="-1.5,0,0,0" >
       <TextBlock
         Text="{Binding RelativeSource={RelativeSource Self},
         Path=Parent.Parent.DataPoint.X,StringFormat='x=0.0;x=-0.0'}"
/>
     </Border>
   </c1chart:ChartPanelObject>
```

The property Attach allows to stick the possible positions of the element to the nearest data point. It can be attached to the single coordinate(X or Y) or both coordintates(XY).

Chart View

The ChartView object represents the area of the chart that contains data (excluding the titles and legend, but including the axes). The View property returns a ChartView object with the following main properties:

Property	Description
Axes	Gets the axis collection. Stores x, y, and z axes. These axes are responsible for the chart range (minimum, maximum, unit, and linear/logarithmic scale) and the appearance of the axis lines, grid lines, tick marks and axis labels.
AxisSize	Gets or sets the relative size of axis area comparing with the whole plot cube.
AxisX, AxisY, AxisZ	Each of these properties returns Axis objects that allow you to customize the appearance of the chart axes.
Margin	Returns a **Margin** object that allows you to specify the distance between the chart area and the plot area. The axes labels are displayed in this space.
PlotRect	Returns a **Rect** object that controls the appearance of the area inside the axes.

ChartView.Style	Contains properties that set the color and border of the chart area.

The following properties in the ChartView class are only applicable in 3D Charts:

Property	Description
Camera	Gets or sets the camera for 3D only.
Lights	Specifies the light source that is used in the scene of 3D charts such as Ambient light, Directional light, and so on.
Perspective	Gets or sets the value of perspective transformation.
Margin	Returns a **Margin** object that allows you to specify the distance between the chart area and the plot area. The axes labels are displayed in this space.
Ratio	Gets or sets the ratio of axes in plot cube.
Transform	Enables you to specify all 3D transformations, including translation, rotation, and scale transformations.

Axes

The axes are represented by sub-properties of the View property: AxisX, AxisY, and AxisZ. Each of these properties returns an **Axis** object with the following main properties:

- Layout, Style, and Value properties

  The following properties below represent the layout and style of the axes in **C1Chart**:

Property	Description
Position	Allows you to set the position of the axis. For example, you may want to display the X-axis above the data instead of below. For more information see Axis Position (page 101).
Reversed	Allows you to reverse the direction of the axis. For example, you can show Y values going down instead of up. For more information see Inverted and Reversed Chart Axes (page 105).
Title	Sets a string to display next to the axis (this is typically used to describe the variable and units being depicted by the axis). For more information see Axis Title (page 101).
Foreground	Gets or sets the foreground brush of the axis.
AxisLine	Gets or sets the axis line. The axis line connects the points on the plot that correspond to the Min and Max of the axis.
IsTime	Gets or sets whether the axis represents time values.
Scale	Gets or sets the scale of the axis.
MinScale	Gets or sets the minimal scale of the axis.

- Annotation properties

  The following properties below represent the format for the annotation of the axes in **C1Chart**:

Property	Description
ItemsSource	Gets or sets the source for axis annotations.
AnnoFormat	A set of predefined formats used to format the values displayed next to the axis.
AnnoAngle	Allows you to rotate the values so they take up less space along the axis. For more information see Axis Annotation Rotation (page 112).
AnnoTemplate	Gets or sets the template for axis annotation.

- Scaling Tickmark and Gridline properties

  The following properties represent the scaling, tickmarks, and gridline styles and function for the axes in **C1Chart**:

Property	Description
AutoMin, AutoMax	Determine whether the minimum and maximum values for the axis should be calculated automatically. For more information see Axis Bounds.
Min, Max	Set the minimum and maximum values for the axis (when AutoMin and AutoMax are set to False). For more information see Axis Bounds (page 104).
MajorUnit, MinorUnit	Set the spacing between the major and minor tickmarks (when the AutoMajor and AutoMinor properties are set to False).
MajorGridFill	Gets or sets the fill based of the major grid. The MajorGridFill enables you to create a striped plot appearance.
MajorGridStroke, MinorGridStroke	Gets or sets the brush of the major/minor grid lines.
MajorGridStrokeDashes, MinorGridStrokeDashes	Gets or sets the dash pattern of the major/minor grid lines.
MajorGridStrokeThickness, MinorGridStrokeThickness	Gets or sets the thickness of the major/minor grid lines.
MajorTickHeight, MinorTickHeight	Gets or sets the major/minor tick height.
MajorTickStroke, MinorTickStroke	Gets or sets the major/minor tick stroke.
MajorTickThickness, MinorTickThickness	Gets or sets the major/minor tick thickness.

Axis Lines

The axis lines are lines that appear horizontally from the starting value to the ending value for the Y-Axis and vertically from the starting value to the ending value for the X-Axis. The Z-axis line is used in 3D charts.

You can use either the **Axis.Foreground** or the **ShapeStyle.Stroke** property to apply color to the axis line. Note that the **Axis.Foreground** property overrides the **ShapeStyle.Stroke** property.

You can specify the type of shape for the starting and ending points of the axis line using the **StrokeStartLineCap** and **StrokeEndLineCap** properties.

Property	Description
Foreground	Gets or sets the foreground brush of the axis.
Stroke	Gets or sets the stroke brush of the shape.
StrokeThickness	Gets or sets the stroke thickness of the shape.
ShapeStyle.StrokeStartLineCap	Gets or sets the shape for the start of the stroke line cap.
ShapeStyle.StrokeEndLineCap	Gets or sets the shape for the end of the stroke line cap.

Dependent Axis

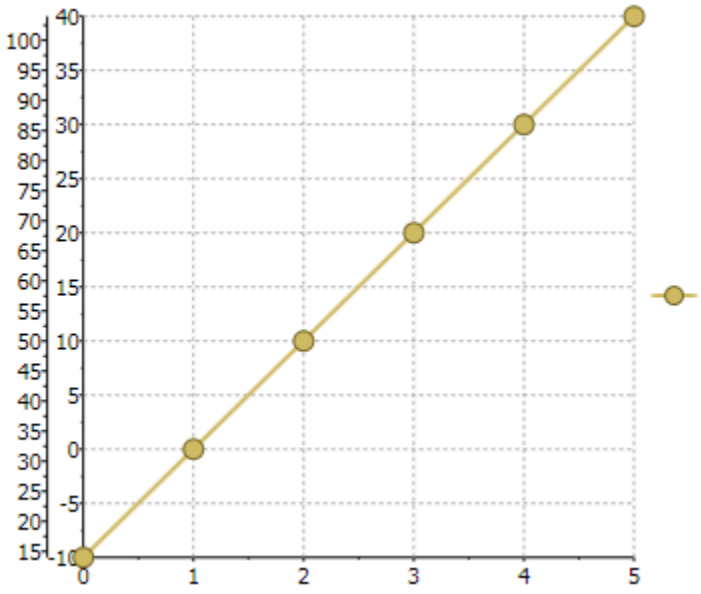
The IsDependent allows to link the auxiliary axis with one from the main axes (AxisX or AxisY, depending on AxisType). The dependent axis always has the same minimum and maximum as the main axis.

New property DependentAxisConverter and delegate Axis.AxisConverter specifies function that is used to convert coordinate from main axis to the dependent axis.

The following code creates a depepdent Y-Axis:

```
c1Chart1.Reset(true);
        c1Chart1.Data.Children.Add(
          new DataSeries() { ValuesSource = new double[] { -10, 0, 10,
20, 30, 40 } });
        c1Chart1.ChartType = ChartType.LineSymbols;
        Axis axis = new Axis() { AxisType = AxisType.Y, IsDependent
=true};
// Celsius -> Fahrenheit
        axis.DependentAxisConverter = (val) => val * 9 / 5 + 32;
        c1Chart1.View.Axes.Add(axis);
```

The following image displays the dependent (leftmost) Y-Axis that shows values in Fahrenheit corresponding to the Celsius on the main Y-axis:



Axis Position

You can specify the axis position by setting the Position property to near or far values. For vertical axis **Axis.Position.Near** corresponds to the left and **Axis.Position.Far** corresponds to the right. For horizontal axis **Axis.Position.Near** corresponds to bottom and **Axis.Position.Far** corresponds to the top.

Axis Title

Adding a title to an axis clarifies what is charted along that axis. For example if your data includes measurements it's helpful to include the unit of measurement (grams, meters, liters, etc) in the axis title. Axis titles can be added to **Area**, **XY-Plot**, **Bar**, **HiLoOpenClose** or **Candle** charts.

The axis titles are **UIElement** objects rather than simple text. This means you have complete flexibility over the format of the titles. In fact, you could use complex elements with buttons, tables, or images for the axis titles.

To set the Axis Title programmatically

```
// Set axes titles
   c1Chart1.View.AxisY.Title= new TextBlock() { Text = "Kitchen
Electronics" };
   c1Chart1.View.AxisX.Title = new TextBlock() { Text = "Price" };
```

To set the Axis Title using XAML code

```
<c1chart:C1Chart >
      <c1chart:C1Chart.View>
        <c1chart:ChartView>
          <c1chart:ChartView.AxisX>
            <c1chart:Axis>
              <c1chart:Axis.Title>
                <TextBlock Text="Price" TextAlignment="Center"
Foreground="Crimson"/>
              </c1chart:Axis.Title>
            </c1chart:Axis>
          </c1chart:ChartView.AxisX>
          <c1chart:ChartView.AxisY>
            <c1chart:Axis>
              <c1chart:Axis.Title>
```

```
                <TextBlock Text="Kitchen Electronics"
TextAlignment="Center" Foreground="Crimson"/>
              </c1chart:Axis.Title>
            </c1chart:Axis>
          </c1chart:ChartView.AxisY>
        </c1chart:ChartView>
      </c1chart:C1Chart.View>
    </c1chart:C1Chart>
```

Axis Tick Marks

The chart automatically sets up the axis with both major and minor ticks. Customizing the tick spacing or attributes is as easy as manipulating a set of properties.

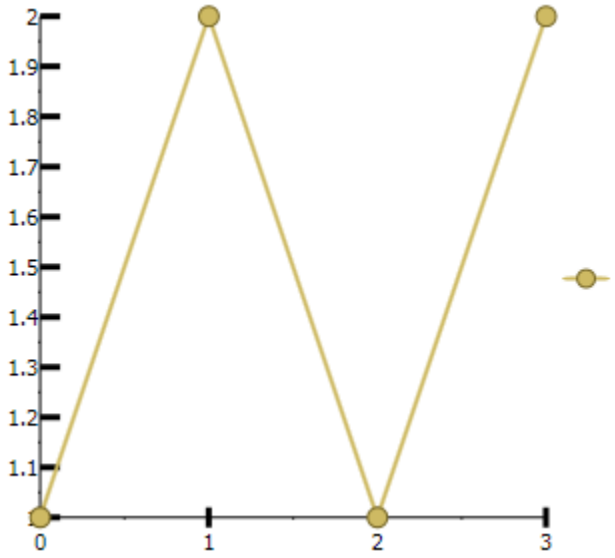
The MajorUnit and MinorUnit properties set the state of the Axis' tick marks. To eliminate clutter in a chart, you can display fewer labels or tick marks on the category (x) axis by specifying the intervals at which you want the categories to be labeled, or by specifying the number of categories that you want to display between tick marks.

Major Tick Overlap

You can determine the overlap value for the major tick mark by specifying a value range from 0 to 1 for the MajorTickOverlap property. The default value is 0, which means there is no overlap. When the overlap is 1, the whole tick is inside the plot area. As you increase the MajorTickOverlap value for the X-Axis, the tick mark moves up and down as you decrease the value. As you increase the MajorTickOverlap value for the Y-Axis the tick mark moves to the left.

```
c1Chart1.Reset(true);
c1Chart1.Data.Children.Add(
new DataSeries() { ValuesSource = new double[] { 1, 2, 1, 2 } });
c1Chart1.ChartType = ChartType.LineSymbols;
c1Chart1.View.AxisX.MajorGridStrokeThickness = 0;
c1Chart1.View.AxisX.MajorTickThickness = 3;
c1Chart1.View.AxisX.MajorTickHeight = 10;
c1Chart1.View.AxisX.MajorTickOverlap = 0;
c1Chart1.View.AxisY.MajorGridStrokeThickness = 0;
c1Chart1.View.AxisY.MajorTickThickness = 3;
c1Chart1.View.AxisY.MajorTickHeight = 10;
c1Chart1.View.AxisY.MajorTickOverlap = 0;
```

The following image displays the MajorTickOverlap value as zero:

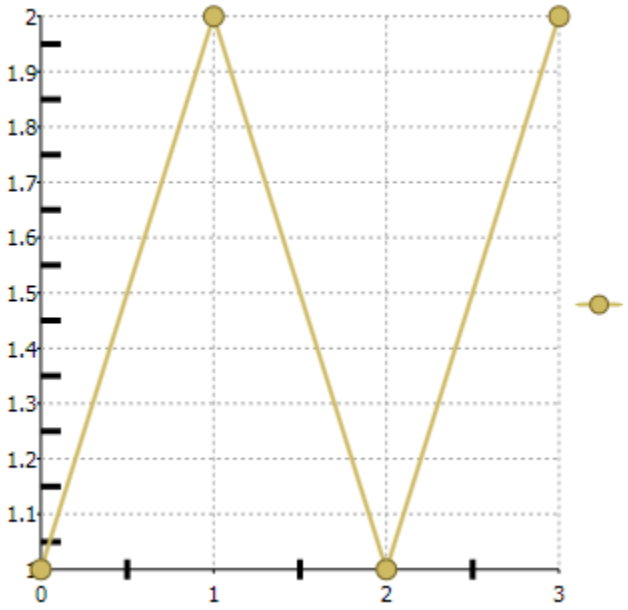


Minor Tick Overlap

You can determine the overlap value for the minor tick mark by specifying a value range from 0 to 1 for the MinorTickOverlap property. The default value is 0 which, means there is no overlap. When the overlap is 1, the whole tick is inside the plot area.

```
c1Chart1.Reset(true);

c1Chart1.Data.Children.Add(
new DataSeries() { ValuesSource = new double[] { 1, 2, 1, 2 } });

c1Chart1.ChartType = ChartType.LineSymbols;
c1Chart1.View.AxisX.MinorGridStrokeThickness = 0;
c1Chart1.View.AxisX.MinorTickThickness = 3;
c1Chart1.View.AxisX.MinorTickHeight = 10;
c1Chart1.View.AxisX.MinorTickOverlap = .5;

c1Chart1.View.AxisY.MinorGridStrokeThickness = 0;
c1Chart1.View.AxisY.MinorTickThickness = 3;
c1Chart1.View.AxisY.MinorTickHeight = 10;
c1Chart1.View.AxisY.MinorTickOverlap = 1;
```

The following image depicts a **MinorTickOverlap** set to "1":



Axis Grid Lines

Grid lines are lines that appear perpendicular with major/minor tick marks at unit major/minor intervals. Grid lines can help improve the readability of the Chart when you are looking for exact values.

To paint or fill the major/minor grid lines

You can apply a color to major/minor grid lines using the MajorGridStroke/MinorGridStroke properties. A fill color can be applied in between each value of the major grid lines using the MajorGridFill properties.

To set the dash pattern for major/minor grid lines

You can set the dash pattern for major/minor grind lines using the MajorGridStrokeDashes/MinorGridStrokeDashes property.

To set the thickness for major/minor grid lines

You can specifiy the thickness for major/minor grid lines using the MajorGridStrokeThickness/MinorGridStrokeThickness properties.

To set the fill for major grid lines

You can apply a fill for the major gridlines using the MajorGridFill property.

Axis Bounds

Normally a graph displays all of the data it contains. However, a specific part of the chart can be displayed by fixing the axis bounds.

The chart determines the extent of each axis by considering the lowest and highest data value and the numbering increment. Setting the Min and Max, AutoMin, and AutoMax properties allows the customization of this process.

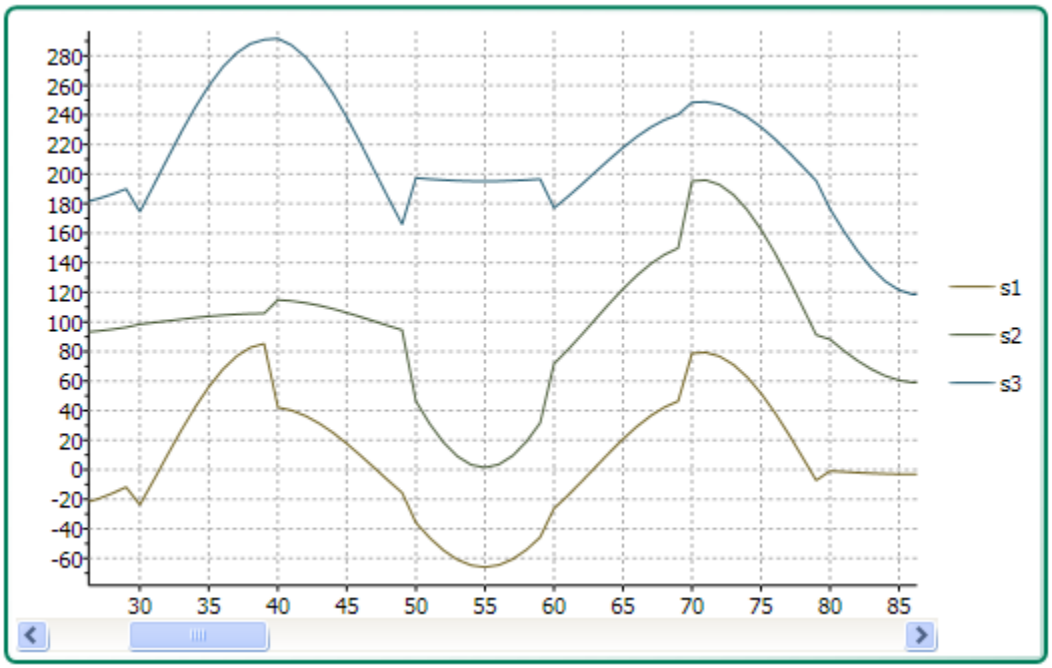
Axis Min and Max

Use the Min and Max properties to frame a chart at specific axis values. If the chart has X-axis values ranging from 0 to 100, then setting Min to 0 and Max to 10 will only display the values up to 10.

The chart can also calculate the Min and Max values automatically. If the AutoMax and AutoMin properties are set to **True** then the chart automatically formats the axis numbering to fit the current data set.

Axis Scrolling

In circumstances when you have a substantial amount of X-values or Y-values in your chart data, you can add an AxisScrollBar control to the axes on your chart. Adding a scrollbar can make the data on the chart easier to read by scrolling through it so you can closely view pieces of data one at a time. The following image has the ScrollBar set to the **View.AxisX.Value** property.



A scrollbar can appear on the X-Axis or Y-Axis simply by setting the ScrollBar's **Value** property to AxisX for the X-Axis or AxisY for the Y-Axis.

The following XAML code shows how to assign a horizontal scrollbar to the X-Axis:

```
<c1chart:C1Chart Name="c1Chart1">
    <c1chart:C1Chart.View>
      <c1chart:ChartView>
        <c1chart:ChartView.AxisX>
          <c1chart:Axis Scale="0.2">
            <c1chart:Axis.ScrollBar>
              <c1chart:AxisScrollBar />
            </c1chart:Axis.ScrollBar>
          </c1chart:Axis>
        </c1chart:ChartView.AxisX>
      </c1chart:ChartView>
    </c1chart:C1Chart.View>
```

Setting the minimum and maximum values for the Scrollbar will prevent the scrollbar from changing the Axis values when you are scrolling.

Inverted and Reversed Chart Axes

When a data set contains X or Y values which span a large range, sometimes the normal chart setup does not display the information most effectively. Formatting a chart with a vertical Y-axis and axis annotation that begins

at the minimum value can sometimes be more visually appealing if the chart could be inverted or the axes reversed. Therefore, C1Chart provides the Inverted property and the Reversed property of the axis.

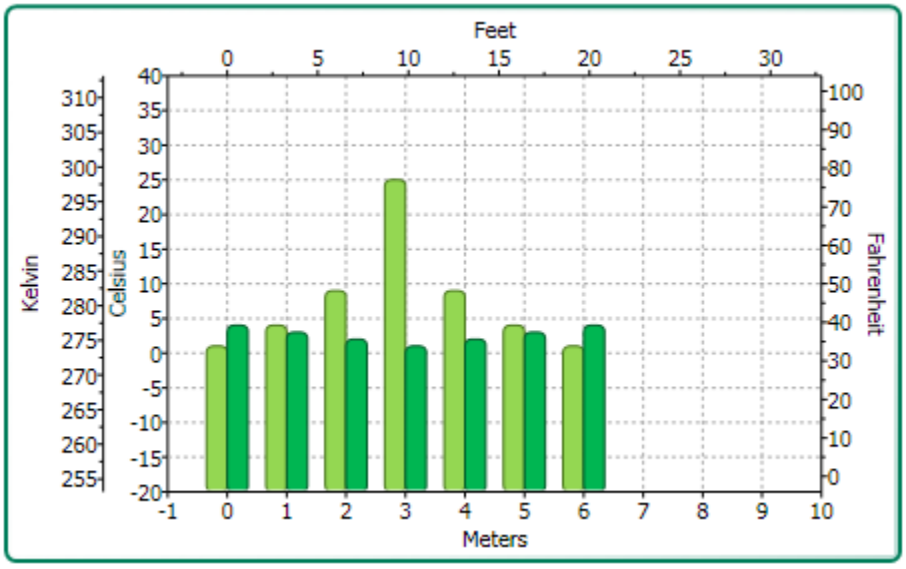
Setting the Reversed property of the ChartView to **True** will reverse the axes. This means that the Max side of the axis will take the place of the Min side of the axis, and the Min side of the axis will take the place of the Max side of the axis. Initially, the chart displays the Minimum value on the left side of the X-axis, and on the bottom side of the Y-axis. Setting the Reversed property of the Axis, however will juxtapose these Maximum and Minimum values.

Multiple Axes

Multiple axes are commonly used when you have the following:

- Two or more Data Series that have mixed types of data which make the scales very different
- Wide range of data values that vary from Data Series to Data Series

The following chart uses five axes to effectively display the length and temperature in both metric and non-metric measurements:



You can add multiple axes to the chart by adding a new Axis object and then specifying its type (X, Y, or Z) for the AxisType property.

The following XAML code shows how to add multiple Y-axes to the chart:

```
<c1chart:C1Chart Margin="0" Name="c1Chart1">
  <c1chart:C1Chart.View>
    <c1chart:ChartView>
      <!-- Auxiliary y-axes -->
      <c1chart:Axis Name="ay2" AxisType="Y" Position="Far" Min="0"
Max="10" />
      <c1chart:Axis Name="ay3" AxisType="Y" Position="Far" Min="0"
Max="20" />
      <c1chart:Axis Name="ay4" AxisType="Y" Position="Far" Min="0"
Max="50" />
    </c1chart:ChartView>
  </c1chart:C1Chart.View>
  <c1chart:C1Chart.Data>
    <c1chart:ChartData>
      <c1chart:DataSeries Values="1 2 3 4 5" />
```

```
      <c1chart:DataSeries AxisY="ay2" Values="1 2 3 4 5" />
      <c1chart:DataSeries AxisY="ay3" Values="1 2 3 4 5" />
      <c1chart:DataSeries AxisY="ay4" Values="1 2 3 4 5" />
    </c1chart:ChartData>
  </c1chart:C1Chart.Data>
</c1chart:C1Chart>
```

Axis Logarithmic Scaling

When data is shown with large differences in scale or when data is expected to vary exponentially on the same chart, it is often convenient to use logarithmic scaling on one or more axes. On a logarithmic axis, equal distance along it depicts an equal percentage change. If logarithmic scaling is used on one or both axes, the chart is called a Log Scale chart.

With logarithmic scaling, values are physically spaced based upon the logarithm of the value instead of the value itself. This is useful when quantities are charted over a very wide range, and when depiction of geometric and/or exponential relationships is desired.

Unlike **arithmetic charts** where changes are measured in terms of direct units, **log scale charts** show changes in terms of percentage change. For example, in a **log scale chart** measuring dollars, a change from $1 to $2 is a 100 percent change so the distance on the chart axis from $1 to $2 would be the same from $50 to $100. Whereas in an **arithmetic chart**, a change from $50 to $100 would make the distance on the axis from $50 to $100 appear much larger on the chart because it is a change of $50 as opposed to just $1.

Commonly Used Logarithms

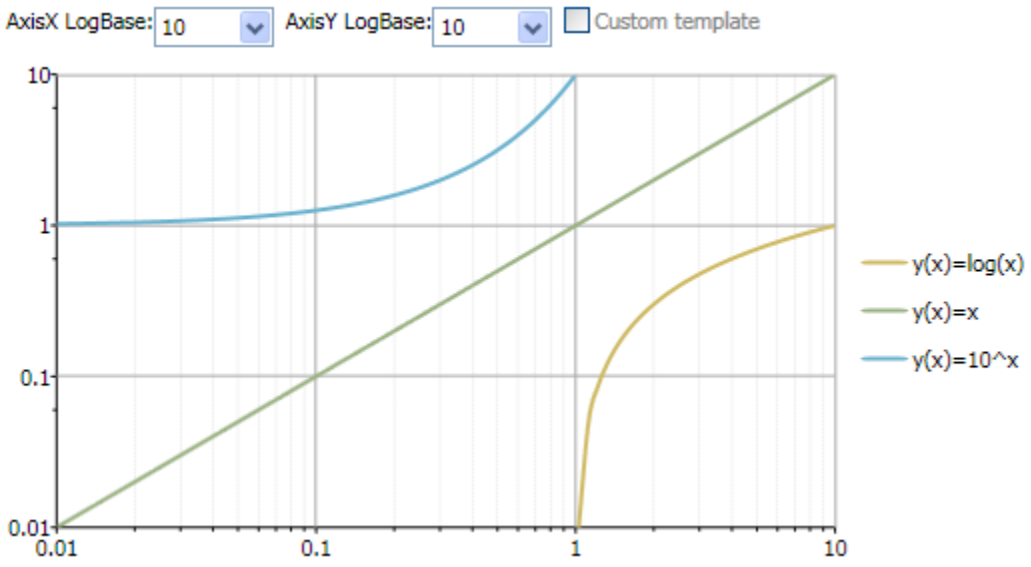
Logarithms can be expressed using any base value, including integers and floating point values. The two most commonly used types of logarithms include:

- **Common logarithms** – Use 10 as the base so it's written as log 100 = 2.
- **Natural logarithms** –Use the mathematical constant e as the base.

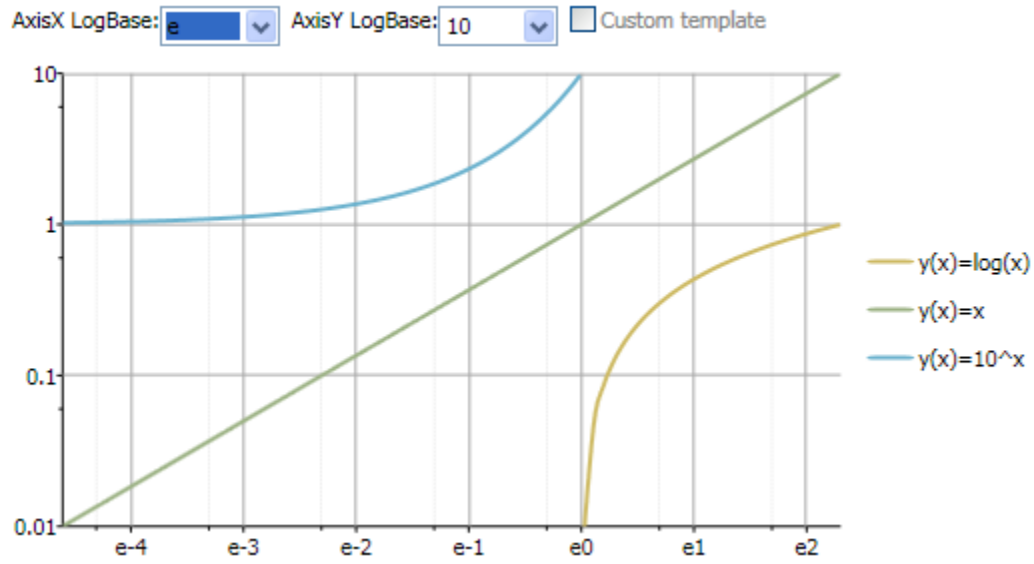
Logarithmic Base

You can specify the value for the logarithmic base using the LogBase property. The default value is double.Nan which corresponds to the default linear axis. A natural logarithm is the logarithm to the base e. Note that Logarithmic scaling does not make mathematical sense when values are less than or equal to zero.

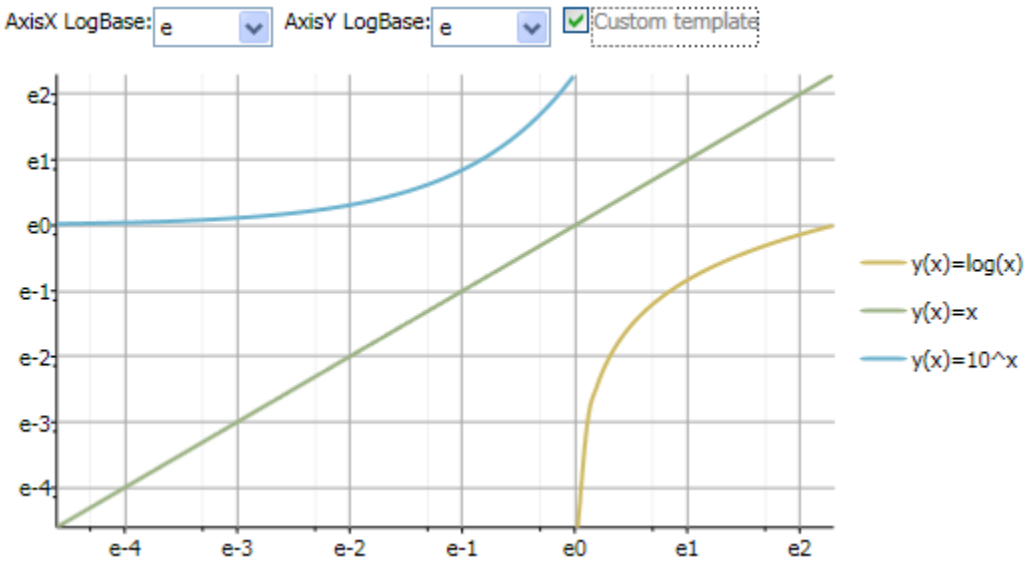
The following image shows how **C1Chart** appears when the LogBase is set to 10 for the X-Axis and Y-Axis, which is the common logarithm:



The following image shows how **C1Chart** appears when the LogBase for the X-Axis is e and the LogBase for the Y-Axis is 10:



The following image shows how **C1Chart** appears when the LogBase for the X-Axis is e and the LogBase for the Y-Axis is e:



Criteria used for Logarithmic Scaling

The following additional criteria must be following for logarithmic axes:

- Any data that is less than or equal to zero is not graphed (it is treated as a data hole), since a logarithmic axis only handles data values that are greater than zero. For the same reason, axis and data minimum/maximum bounds and origin properties cannot be set to zero or less.
- Axis numbering increment, ticking increment, and precision properties have no effect when the axis is logarithmic.
- For a logarithmic X-axis, the chart type must be either plot, bubble, area, HiLo, HiLoOpenClose or candle. For the Y-axis, the chart type must be either plot, bubble, area, polar, HiLo, HiLoOpenClose, candle, radar or filled radar.

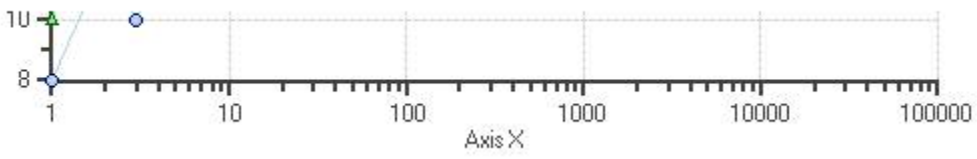
UnitMajor and Logarithmic Axes

For logarithmic axis scaling, MajorUnit is taken as a multiplier of the base value of each cycle that provides a hint as to the annotation spacing within each cycle of the logarithmic base. That is (MajorUnit * base cycle value) is approximately the annotation value increment within each cycle. For integer logarithmic base values, the result is usually exact. For floating point values, annotations are rounded to nice numbers as for linear scaling.

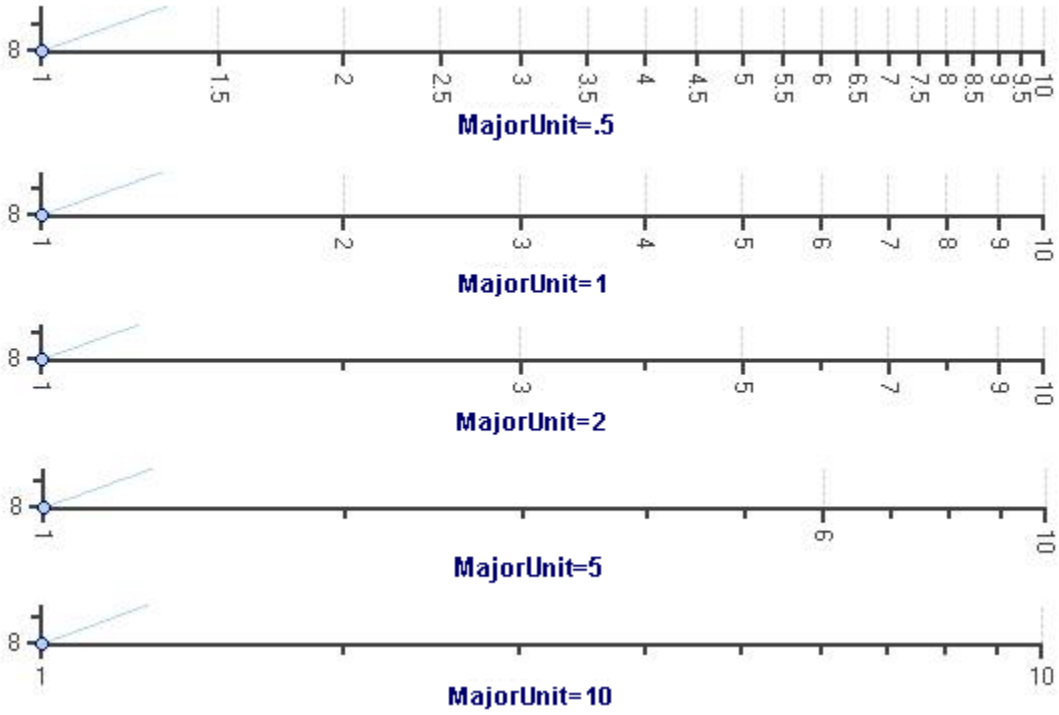
Detailed Explanation of UnitMajor and Logarithmic Axes

Often, when logarithmic scales are used, the bounds of a chart axis will span multiple cycles of the logarithmic base. In these cases, the usual linear specification of MajorUnit no longer makes sense, as a value appropriate for a given cycle makes little sense for the previous or next cycles. For the MajorUnit setting to be of value, it must pertain to values relative to each cycle of the logarithmic base.

If this doesn't make sense to you, think about what single, fixed, incremental value you might use for the following axis:



Following the above reasoning, for logarithmic axes, the chart assumes that MajorUnit specifies the fraction of the base value for each cycle. Consider the following examples:



In each case, the base cycle value is 1. For each cycle the next annotation value = previous number + (base value of cycle * MajorUnit). The maximum value of the MajorUnit is the LogarithmicBase. The automatic value of MajorUnit is always the LogBase.

When all of the annotation values are calculated, a nice rounding algorithm is applied so the numbers are relatively easy to read. The behavior may seem a bit odd, but it is the result of accommodating any logarithmic base while at the same time obtaining numbers for the annotations that are reasonable to read.

For example, the plots above are log-base 10 values, but there are also natural-logs to consider such as log-base 2, log-base-x, etc.

Axes Annotation

The annotation along each axis is an important part of any chart. The chart annotates the axes with numbers based on the data/values entered into the BubbleSeries, DataSeries, HighLowOpenCloseSeries, HighLowSeries, or XYDataSeries objects. Annotation for the Axes will always display basic text without any formatting applied to them.

The chart automatically produces the most natural annotation possible, even as chart data changes. The following Annotation properties can be modified to perfect this process:

Property	Description
AnnoFormat	A set of predefined formats used to format the values displayed next to the axis.

AnnoAngle	Gets or sets the rotation angle of axis annotation. This allows you to rotate the values so they will take up less space along the axis.
AnnoTemplate	Gets or sets the template for the axis annotation. This is useful for building custom annotations. For an example, see Creating a Custom Annotation (page 159).
ItemsSource	Gets or sets the source for axis annotations.

Axis Annotation Format

You can control the annotation formatting for the values on the X or Y axis using the AnnoFormat property.

Setting the AnnoFormat property to a .NET Framework composite format string will format the data entered into the property. For more information on the standard numeric format strings that you can use for the AnnoFormat property see Standard Numeric Format Strings.

DateTime Format Strings

The DateTime format strings are divided into two categories:

- Standard Date Time Format Strings
- Custom Date Time Format Strings

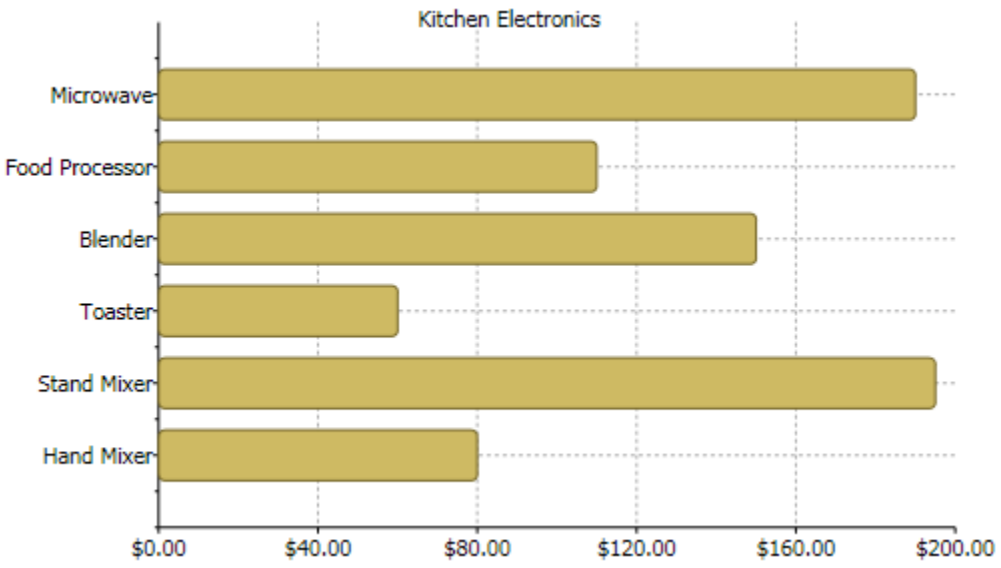
Numeric Format Strings

- Standard Numeric Format Strings
- Custom Numeric Format Strings

Custom Numeric Format Strings

You can also customize your format strings by using the custom numeric format strings.

To use the AnnoFormat property specify a standard or custom format string for it. For example the following Bar chart's AnnoFormat property is set to "c" to change the whole values to currency format.



XAML

```
<c1chart:C1Chart.View>
                <c1chart:ChartView>
```

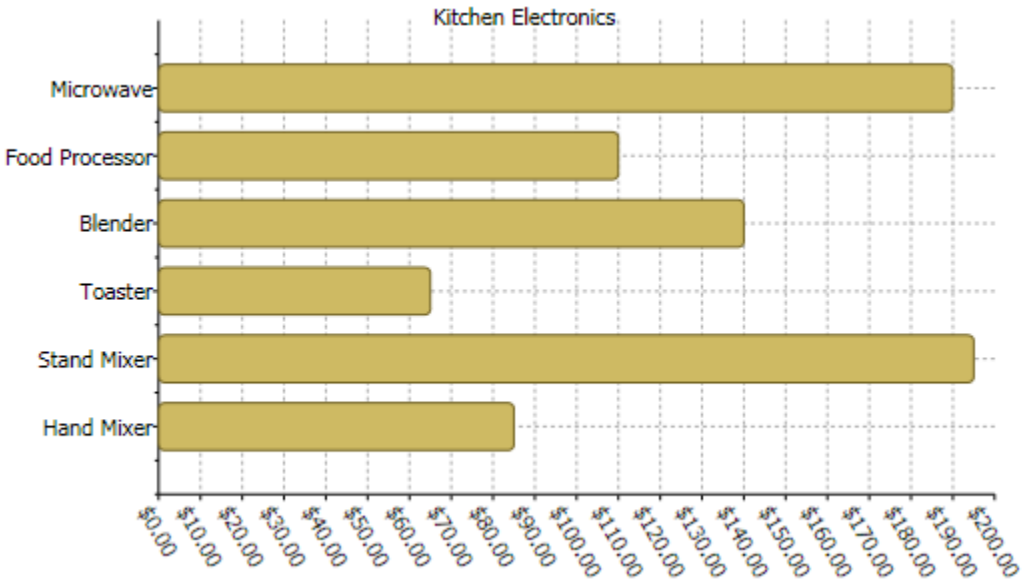
```
                    <c1chart:ChartView.AxisX>
                        <c1chart:Axis Min="0" AnnoFormat="c"
AutoMin="false" AutoMax="false" Max="200" />
                    </c1chart:ChartView.AxisX>
                </c1chart:ChartView>
            </c1chart:C1Chart.View>
```

C#

```
  // Financial formatting
   c1Chart1.View.AxisX.AnnoFormat = "c";
   c1Chart1.View.AxisX.Min = 0;
```

Axis Annotation Rotation

Use the AnnoAngle property to rotate the axis annotation counterclockwise from the specified number of degrees. This property is especially useful if the X-axis is crowded with annotation. Rotating the annotations +/- 30 or 60 degrees allows a much larger number of annotations in a confined space on horizontal axes. By utilizing the AnnoAngle property, the X-axis annotation does not overlap, as shown below:



XAML

```
<c1chart:C1Chart.View>
                <c1chart:ChartView>
                    <c1chart:ChartView.AxisX>
                        <c1chart:Axis Min="0" MajorUnit="10"
AnnoFormat="c" AutoMin="false" AutoMax="false" Max="200" AnnoAngle="60"
/>
                    </c1chart:ChartView.AxisX>
                </c1chart:ChartView>
            </c1chart:C1Chart.View>
```

C#

```
// Financial formatting
c1Chart1.View.AxisX.AnnoFormat = "c";
c1Chart1.View.AxisX.Min = 0;
c1Chart1.View.AxisX.AnnoAngle = "60";
```

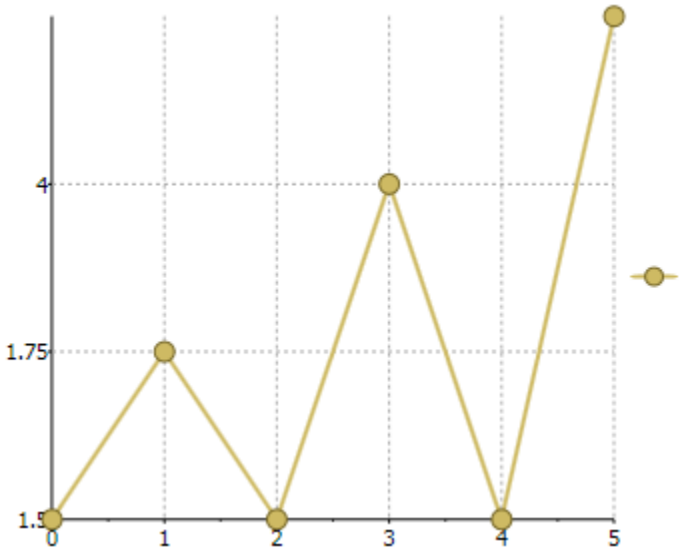
Custom Axis Annotation

In some situations you may need to create custom axis annotation. The following scenarios can be useful for creating custom axis annotation:

- When the ItemsSource property is a collection of numbers or **DateTime** values the chart uses these values as axis labels. The following code uses the ItemsSource property to create the custom Y-axis labels:

```
c1Chart1.Reset(true);
        c1Chart1.Data.Children.Add(
          new DataSeries() { ValuesSource = new double[] { 1, 2, 1, 3, 1,
4 } });
        c1Chart1.ChartType = ChartType.LineSymbols;
        c1Chart1.View.AxisY.ItemsSource = new double[] { 1.25, 1.5, 1.75,
4 };
```

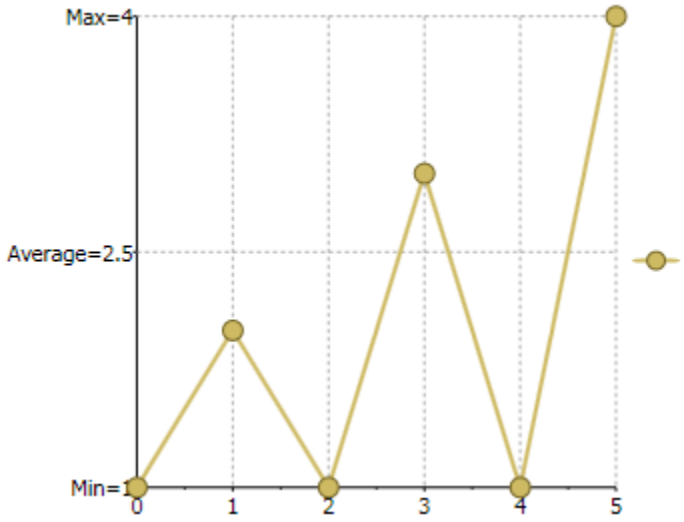
  Here is what the chart appears like after adding the preceding code:



- When the ItemsSource property is a collection of **KeyValuePair**<object, double> or **KeyValuePair**<object, DateTime> the chart creates axis labels based on the provided pairs of values. For example, the following code uses the **KeyValuePair** to create the custom axis annotation for the Y axis:

```
c1Chart1.Reset(true);

        c1Chart1.Data.Children.Add(
          new DataSeries() { ValuesSource = new double[] { 1, 2, 1, 3, 1,
4 } });
        c1Chart1.ChartType = ChartType.LineSymbols;

        c1Chart1.View.AxisY.ItemsSource = new
List<KeyValuePair<object,double>>
          { new KeyValuePair<object,double>("Min=1", 1),
            new KeyValuePair<object,double>("Average=2.5", 2.5),
            new KeyValuePair<object,double>("Max=4", 4)};
```

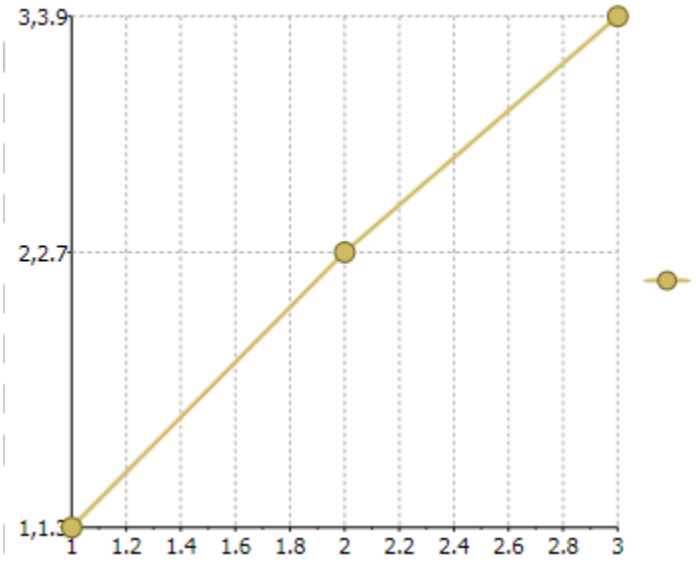
  Here is what the chart appears like after adding the preceding code:



- You can use the ItemsValueBinding and ItemsLabelBinding properties to create axis labels using arbitrary collection as data source, like in the following code:

```
   c1Chart1.Reset(true);

        Point[] pts = new Point[] { new Point(1, 1.3), new Point(2, 2.7),
new Point(3, 3.9) };
        c1Chart1.DataContext = pts;
        c1Chart1.ChartType = ChartType.LineSymbols;

        c1Chart1.View.AxisY.ItemsSource = pts;
        c1Chart1.View.AxisY.ItemsValueBinding = new Binding("Y");
        c1Chart1.View.AxisY.ItemsLabelBinding = new Binding();
```

Here is what the chart appears like after adding the preceding code:

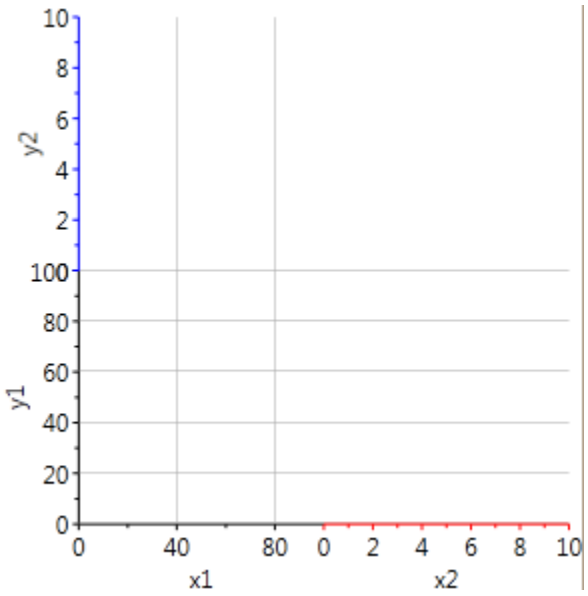


Plot Area

The data is plotted in the plot area of the chart. The Plot area is the part of the plot limited by axes and containing all plot elements (bars, columns, lines etc.). Previously, the chart can have only one plot area but now it's possible to have several ones in the same chart.

Usually plot areas are created automatically based on PlotareaIndex property. By default it's 0 and the new plot area is not created for the additional axis. Axis is just added for example, at the left of main y-axis or at the bottom of the main x-axis. But if you set PlotAreaIndex = 1 the new axis is added on the same line as the main axis. For x-axis the auxiliary axis will be at the right and for y-axis - at the top.

The following example illustrates the new axis added on the same line as the main axis:




```
<c1chart:C1Chart x:Name="chart" >
  <c1chart:C1Chart.View>
    <c1chart:ChartView>
      <!-- Main axes  -->
      <c1chart:ChartView.AxisX>
        <c1chart:Axis Min="0" Max="100" Title="x1" />
      </c1chart:ChartView.AxisX>
      <c1chart:ChartView.AxisY>
        <c1chart:Axis Min="0" Max="100" Title="y1" />
      </c1chart:ChartView.AxisY>

      <!-- Auxiliary axis at the right of main x-axis -->
      <c1chart:Axis x:Name="x2" Title="x2" PlotAreaIndex="1"
         AxisType="X" Min="0" Max="10" />

      <!-- Auxiliary axis at the top of main x-axis -->
      <c1chart:Axis x:Name="y2" Title="y2" PlotAreaIndex="1"
         AxisType="Y" Min="0" Max="10" />

    </c1chart:ChartView>
  </c1chart:C1Chart.View>
</c1chart:C1Chart>
```

To add the data you need to specify the name of the axis (**DataSeries.AxisX/AxisY**) and the data will be plotted along the auxiliary axis.

Plot Area Size

The **PlotArea** size can be specified using the **ColumnDefinitions** and **RowDefinitions** collections in the class **PlotAreaCollection**. The approach is similar to working with the standard grid control. The first collection contains column attributes(widths). The second collection is for the row(height). By default, the plot areas have the same width and the same height.

The following examples show how to programatically specify the size of the plot area:

```
    // widths
    // the width of first plot area is default(fill available space)
    chart.View.PlotAreas.ColumnDefinitions.Add(new
PlotAreaColumnDefinition());
    // the width of second plot area is constant 100 px
    chart.View.PlotAreas.ColumnDefinitions.Add(new
PlotAreaColumnDefinition()
      { Width= new GridLength(100) });
    // heights
    // the height of first plot area is 1*
    chart.View.PlotAreas.RowDefinitions.Add(new PlotAreaRowDefinition()
      { Height = new GridLength(1, GridUnitType.Star) });
    // the height of second plot area is 2*
    chart.View.PlotAreas.RowDefinitions.Add(new PlotAreaRowDefinition()
      { Height = new GridLength(2, GridUnitType.Star) });
```

Plot Area Appearance

You can modify the PlotArea's appearance by using the **Background** and **Stroke/StrokeThickness** properties for the border of the plot area. The plot areas are referenced by using the row/column (the same as the elements in the grid).

The following sample shows how to modify the Plot Area appearance:

```
    <c1chart:ChartView.PlotAreas>
      <!-- row=0 col=0 -->
      <c1chart:PlotArea Background="#10FF0000" Stroke="Red" />
      <!-- row=1 col=0 -->
      <c1chart:PlotArea Row="1" Background="#1000FF00" />
      <!-- row=0 col=1 -->
      <c1chart:PlotArea Column="1" Background="#100000FF" />
      <!-- row=1 col=1 -->
      <c1chart:PlotArea Row="1" Column="1" Background="#10FFFF00"
Stroke="Yellow" />
    </c1chart:ChartView.PlotAreas>
```

Data Aggregation

Data aggregation can be used on the entire **C1Chart** control through the Aggregate property or used on individual series through the Aggregate property.

Data aggregation is when data is gathered and is reflected in a summary form. Commonly, aggregation is used to collect more information about specific groups based on certain variables such as geographic location, income, and age.

C1Chart enables you to use aggregate functions for a grouped data by specifying it when the DataSeries is created. For each **DataSeries** you can choose from one of the following functions using the Aggregate enumeration:

Member name	Description
None	Raw values (no aggregation).
Sum	Calculates the sum of all values for each point.
Count	Number of values for each point.
Average	Average of all values for each point.
Minimum	Gets the minimum value for each point.
Maximum	Gets the maximum value for each point.
Variance	Gets the variance of the values for each point (sample).
VariancePop	Gets the variance of the values for each point (population).
StandardDeviation	Gets the standard deviation of the values for each point (sample).
StandardDeviationPop	Gets the standard deviation of the values for each point (population).

Data Labels

Data labels are labels associated with data points on the chart. They can be useful on some charts by making it easier to see which series a particular point belongs to, or its exact value.

C1Chart supports data labels. Each data series has a PointLabelTemplate property that specifies the visual element that should be displayed next to each point. The PointLabelTemplate is usually defined as a XAML resource, and may be assigned to the chart from XAML or from code.

You can add a DataTemplate to determine both visual aspects of how the data is presented and how data binding accesses the presented data.

To define the PointLabelTemplate as a XAML resource you can create a Resource Dictionary, add the DataTemplate resource to your Resource Dictionary and then in your Window.xaml file you can access the DataTemplate resource.

To add a new resource dictionary:

1. In Solution Explorer, right-click your project, point to **Add**, and then select **Resource Dictionary**. The **Add New Item** dialog box appears.
2. In the Name text box, name the dictionary Resources.xaml and click the **Add** button.

   Resources.xaml is added to the project and opens in the code editor.

To create a label you need to create the label template and assign the PointLabelTemplate to the template.

When rendering the plot for each data point the label is created based on the specified template. The **DataContext** property of the label is set to the current **DataPoint** instance that provides information about the point. When using data binding it makes it easier to display this information in the label.

Here is the sample of a label template that displays the value of the point.

```
<DataTemplate x:Key="lbl">
   <TextBlock Text="{Binding Path=Value}" />
</DataTemplate>
```

After you define a resource, you can reference the resource to be used for a property value by using a resource markup extension syntax that specifies the key name

To assign the template to the data series set the PointLabelTemplate property to the following:

```
<c1chart:DataSeries PointLabelTemplate="{StaticResource lbl}" />
```

Since it is a standard data template, the complex label can be built, for example, the next sample template defines the data label for the **XY** chart which shows both coordinates of the data point.

It uses the standard grid with two columns and two rows as a container. The x-value of the point is obtained with indexer of the DataPoint class. The indexer allows getting the values for the data series classes which support several data sets, such as XYDataSeries class.

```
<DataTemplate x:Key="lbl">
   <!-- Grid 2x2 with black border  -->
   <Border BorderBrush="Black">
         <Grid>
               <Grid.ColumnDefinitions>
                     <ColumnDefinition />
                     <ColumnDefinition />
               </Grid.ColumnDefinitions>
               <Grid.RowDefinitions>
                     <RowDefinition />
                     <RowDefinition />
               </Grid.RowDefinitions>
               <!-- x-coordinate -->
               <TextBlock Text="X=" />
               <TextBlock Grid.Column="1" Text="{Binding Path=[XValues]}" />
               <!-- y-coordinate -->
               <TextBlock Grid.Row="1" Text="Y=" />
               <TextBlock Grid.Row="1" Grid.Column="1" Text="{Binding
Path=Value}" />
         </Grid>
   </Border>
</DataTemplate>
```

When displaying the numerical data value often it is necessary to format the output value. With the static class Format you can specify standard .Net format string inside the XAML code. For example, the sample code uses converter to format percentage value.

```
<DataTemplate x:Key="lbl1">
   <TextBlock Text="{Binding Path=PercentageSeries,
         Converter={x:Static c1chart:Converters.Format},
         ConverterParameter=#.#%}"/>
</DataTemplate>
```

Chart Styles

Plot elements support WPF styles that are a convenient way to control the appearance of chart.

MouseOver Style

The following example shows how to create a style that sets the **Stroke** property for a PlotElement to *Black*:

```
<Window.Resources>
   ...
   <Style x:Key="mouseOver" TargetType="{x:Type c1c:PlotElement}">
         <!-- Default black outline -->
         <Setter Property="Stroke" Value="Black" />
```

```
        <Style.Triggers>
        <!-- When mouse is over the element make thick red outline  -->
              <Trigger Property="IsMouseOver" Value="true">
                    <Setter Property="Stroke" Value="Red" />
                    <Setter Property="StrokeThickness" Value="3" />
                    <Setter Property="Canvas.ZIndex" Value="1" />”
              </Trigger>
        </Style.Triggers>
   </Style>
</Window.Resources>
```

Note: When you set the TargetType of your style to the PlotElement type without assigning the style with an x:Key, the style gets applied to both of your PlotElement elements.

To apply the mouseover styles to the data series you can use set the SymbolStyle property like the following:

```
<c1c:DataSeries … SymbolStyle="{StaticResource mouseOver}"/>
```

Chart Appearance

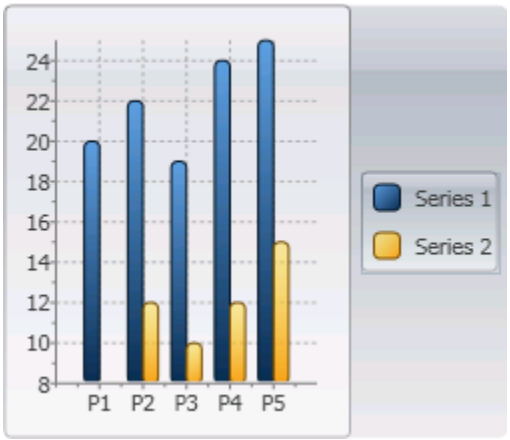
When the chart's data and axes are formatted properly, its elements can be customized to make it look clearer and more professional. The following topics cover tasks that customize the appearance of a chart.

Chart Themes

ComponentOne Chart for WPF incorporates several themes, including **Office 2003 Vista**, and **Office 2007** themes that allow you to customize the appearance of your chart. The built-in themes are described and pictured below:

Office2007Black Theme

This is the default theme based on the **Office 2007 Black** style and it appears as a dark gray-colored chart with orange highlighting.



In XAML

To specifically define the **Office2007Black** theme in your chart add the following `Theme` XAML to the `<c1chart:C1Chart>` tag so that it appears similar to the following:

```
<c1chart:C1Chart Name="c1Chart1" Theme="{DynamicResource
{ComponentResourceKey TypeInTargetAssembly=c1chart:C1Chart,
ResourceId=Office2007Black}}">
```

In Code

To specifically define the **Office2007Black** theme in your chart, add the following code your project:

- Visual Basic

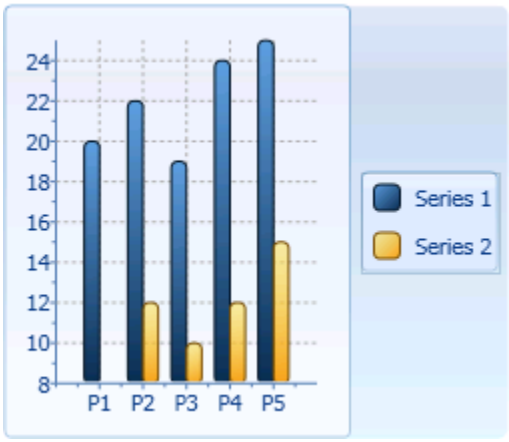
```
C1Chart1.Theme = TryCast(C1Chart1.TryFindResource( _
   New ComponentResourceKey(GetType(C1Chart), "Office2007Black")), _
   ResourceDictionary)
```

- C#

```
c1Chart1.Theme = c1Chart1.TryFindResource(
   new ComponentResourceKey(typeof(C1.WPF.C1Chart.C1Chart),
   "Office2007Black")) as ResourceDictionary;
```

Office2007Blue Theme

This theme is based on the **Office 2007 Blue** style and it appears as a blue-colored chart with orange highlighting.



In XAML

To specifically define the **Office2007Blue** theme in your chart, add the following `Theme` XAML to the `<c1chart:C1Chart>` tag so that it appears similar to the following:

```
<c1chart:C1Chart Name="c1Chart1" Theme="{DynamicResource
{ComponentResourceKey TypeInTargetAssembly=c1chart:C1Chart,
ResourceId=Office2007Blue}}">
```

In Code

To specifically define the **Office2007Blue** theme in your chart, add the following code your project:

- Visual Basic

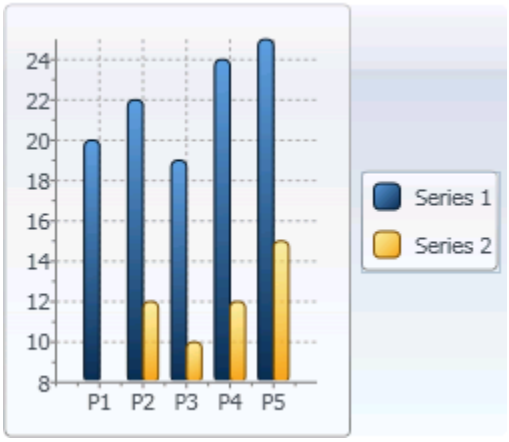
```
C1Chart1.Theme = TryCast(C1Chart1.TryFindResource( _
   New ComponentResourceKey(GetType(C1Chart), " Office2007Blue")), _
   ResourceDictionary)
```

- C#

```
C1Chart1.Theme = c1Chart1.TryFindResource(
   new ComponentResourceKey(typeof(C1.WPF.C1Chart.C1Chart),
   "Office2007Blue")) as ResourceDictionary;
```

Office2007Silver Theme

This theme is based on the **Office 2007 Silver** style and it appears as a silver-colored chart with orange highlighting.



In XAML

To specifically define the **Office2007Silver** theme in your chart, add the following `Theme` XAML to the `<c1chart:C1Chart>` tag so that it appears similar to the following:

```
<c1chart:C1Chart Name="c1Chart1" Theme="{DynamicResource
{ComponentResourceKey TypeInTargetAssembly=c1chart:C1Chart,
ResourceId=Office2007Silver}}">
```

In Code

To specifically define the **Office2007Silver** theme in your chart, add the following code your project:

- Visual Basic

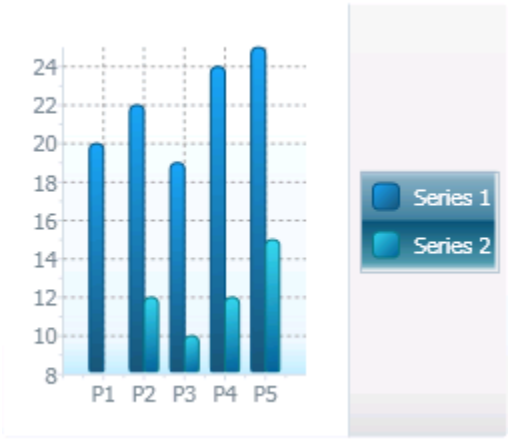
```
C1Chart1.Theme = TryCast(C1Chart1.TryFindResource(
   New ComponentResourceKey(GetType(C1Chart), "Office2007Silver")), _
   ResourceDictionary)
```

- C#

```
c1Chart1.Theme = c1Chart1.TryFindResource(
   new ComponentResourceKey(typeof(C1.WPF.C1Chart.C1Chart),
   "Office2007Silver")) as ResourceDictionary;
```

Vista Theme

This theme is based on the **Vista** style and it appears as a teal-colored chart with blue highlighting.



In XAML

To specifically define the **Vista** theme in your chart, add the following `Theme` XAML to the `<c1chart:C1Chart>` tag so that it appears similar to the following:

```
<c1chart:C1Chart Name="c1Chart1" Theme="{DynamicResource
{ComponentResourceKey TypeInTargetAssembly=c1chart:C1Chart,
ResourceId=Vista}}">
```

In Code

To specifically define the **Vista** theme in your chart, add the following code your project:

- Visual Basic

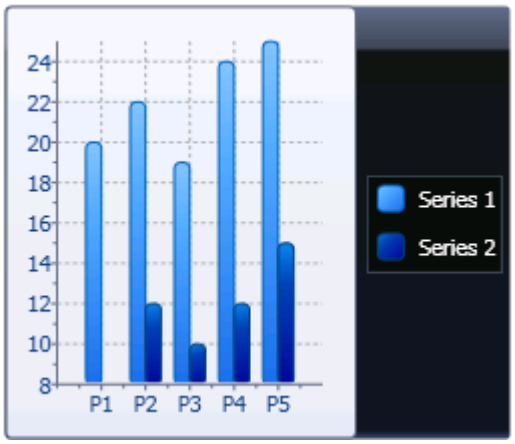
```
C1Chart1.Theme = TryCast(C1Chart1.TryFindResource( _
   New ComponentResourceKey(GetType(C1Chart), "Vista")), _
   ResourceDictionary)
```

- C#

```
c1Chart1.Theme = c1Chart1.TryFindResource(
   new ComponentResourceKey(typeof(C1.WPF.C1Chart.C1Chart),
   "Vista")) as ResourceDictionary;
```

MediaPlayer Theme

This theme is based on the **Windows Media Player** style and it appears as a black-colored chart with blue highlighting.



In XAML

To specifically define the **MediaPlayer** theme in your chart, add the following `Theme` XAML to the `<c1chart:C1Chart>` tag so that it appears similar to the following:

```
<c1chart:C1Chart Name="c1Chart1" Theme="{DynamicResource
{ComponentResourceKey TypeInTargetAssembly=c1chart:C1Chart,
ResourceId=MediaPlayer}}">
```

In Code

To specifically define the **MediaPlayer** theme in your chart, add the following code your project:

- Visual Basic

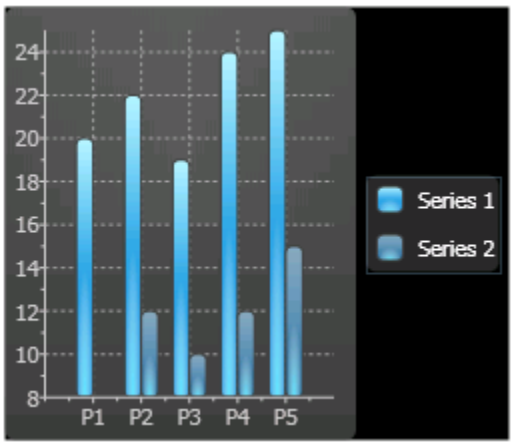
```
C1Chart1.Theme = TryCast(C1Chart1.TryFindResource( _
   New ComponentResourceKey(GetType(C1Chart), "MediaPlayer")), _
   ResourceDictionary)
```

- C#

```
c1Chart1.Theme = c1Chart1.TryFindResource(
   new ComponentResourceKey(typeof(C1.WPF.C1Chart.C1Chart),
   "MediaPlayer")) as ResourceDictionary;
```

DuskBlue Theme

This theme appears as a charcoal-colored chart with electric blue and orange highlighting.



In XAML

To specifically define the **DuskBlue** theme in your chart, add the following `Theme` XAML to the `<c1chart:C1Chart>` tag so that it appears similar to the following:

```
<c1chart:C1Chart Name="c1Chart1" Theme="{DynamicResource
{ComponentResourceKey TypeInTargetAssembly=c1chart:C1Chart,
ResourceId=DuskBlue}}">
```

In Code

To specifically define the **DuskBlue** theme in your chart, add the following code your project:

- Visual Basic

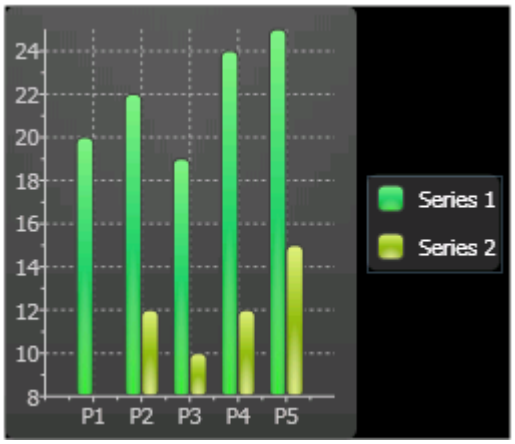
```
C1Chart1.Theme = TryCast(C1Chart1.TryFindResource( _
   New ComponentResourceKey(GetType(C1Chart), "DuskBlue")), _
   ResourceDictionary)
```

- C#

```
c1Chart1.Theme = c1Chart1.TryFindResource(
   new ComponentResourceKey(typeof(C1.WPF.C1Chart.C1Chart),
   "DuskBlue")) as ResourceDictionary;
```

DuskGreen Theme

This theme appears as a charcoal -colored chart with electric green and purple highlighting.



In XAML

To specifically define the **DuskGreen** theme in your chart, add the following `Theme` XAML to the `<c1chart:C1Chart>` tag so that it appears similar to the following:

```
<c1chart:C1Chart Name="c1Chart1" Theme="{DynamicResource
{ComponentResourceKey TypeInTargetAssembly=c1chart:C1Chart,
ResourceId=DuskGreen}}">
```

In Code

To specifically define the **DuskGreen** theme in your chart, add the following code your project:

- Visual Basic

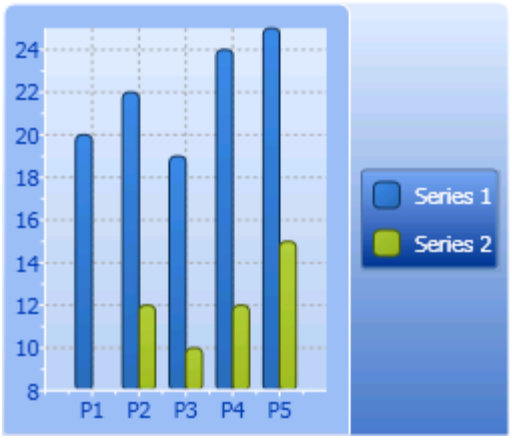
```
C1Chart1.Theme = TryCast(C1Chart1.TryFindResource( _
   New ComponentResourceKey(GetType(C1Chart), "DuskGreen")), _
   ResourceDictionary)
```

- C#

```
c1Chart1.Theme = c1Chart1.TryFindResource(
   new ComponentResourceKey(typeof(C1.WPF.C1Chart.C1Chart),
   " DuskGreen")) as ResourceDictionary;
```

Office2003Blue Theme

This theme is based on the **Office 2003 Blue** style and it appears as a neutral-colored chart with blue and orange highlighting.



In XAML

To specifically define the **Office2003Blue** theme in your chart, add the following `Theme` XAML to the `<c1chart:C1Chart>` tag so that it appears similar to the following:

```
<c1chart:C1Chart Name="c1Chart1" Theme="{DynamicResource
{ComponentResourceKey TypeInTargetAssembly=c1chart:C1Chart,
ResourceId=Office2003Blue}}">
```

In Code

To specifically define the **Office2003Blue** theme in your chart, add the following code your project:

- Visual Basic

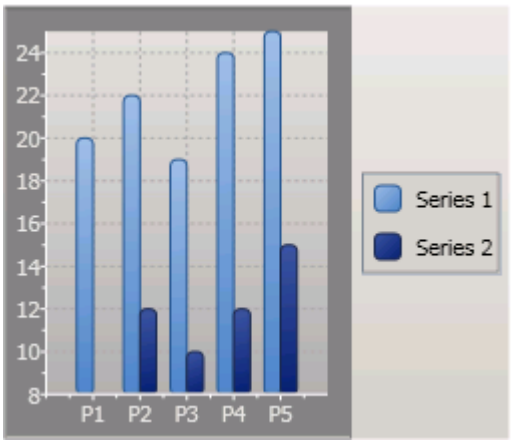
```
C1Chart1.Theme = TryCast(C1Chart1.TryFindResource( _
   New ComponentResourceKey(GetType(C1Chart), "Office2003Blue")), _
   ResourceDictionary)
```

- C#

```
c1Chart1.Theme = c1Chart1.TryFindResource(
   new ComponentResourceKey(typeof(C1.WPF.C1Chart.C1Chart),
   "Office2003Blue")) as ResourceDictionary;
```

Office2003Classic Theme

This theme is based on the **Office 2003 Classic** style and appears as a gray-colored chart with slate-colored highlighting.



In XAML

To specifically define the Office2003Classic theme in your chart, add the following `Theme` XAML to the `<c1chart:C1Chart>` tag so that it appears similar to the following:

```
<c1chart:C1Chart Name="c1Chart1" Theme="{DynamicResource
{ComponentResourceKey TypeInTargetAssembly=c1chart:C1Chart,
ResourceId=Office2003Classic}">
```

In Code

To specifically define the **Office2003Classic** theme in your chart, add the following code your project:

- Visual Basic

```
C1Chart1.Theme = TryCast(C1Chart1.TryFindResource( _
   New ComponentResourceKey(GetType(C1Chart), "Office2003Classic")), _
   ResourceDictionary)
```

- C#

```
c1Chart1.Theme = c1Chart1.TryFindResource(
   new ComponentResourceKey(typeof(C1.WPF.C1Chart.C1Chart),
   "Office2003Classic")) as ResourceDictionary;
```

Office2003Olive Theme

This theme is based on the **Office 2003 Olive** style and it appears as a neutral-colored chart with olive green and orange highlighting.

In XAML

To specifically define the **Office2003Olive** theme in your chart, add the following `Theme` XAML to the `<c1chart:C1Chart>` tag so that it appears similar to the following:

```
<c1chart:C1Chart Name="c1Chart1" Theme="{DynamicResource
{ComponentResourceKey TypeInTargetAssembly=c1chart:C1Chart,
ResourceId=Office2003Olive}}">
```

In Code

To specifically define the **Office2003Olive** theme in your chart, add the following code your project:

- Visual Basic

```
C1Chart1.Theme = TryCast(C1Chart1.TryFindResource( _
   New ComponentResourceKey(GetType(C1Chart), "Office2003Olive")), _
   ResourceDictionary)
```

- C#

```
c1Chart1.Theme = c1Chart1.TryFindResource(
   new ComponentResourceKey(typeof(C1.WPF.C1Chart.C1Chart),
   "Office2003Olive")) as ResourceDictionary;
```

Office2003Royale Theme

This theme is similar to the **Office 2003 Royale** style and appears as a silver-colored chart with blue highlighting.

In XAML

To specifically define the **Office2003Royale** theme in your chart, add the following `Theme` XAML to the `<c1chart:C1Chart>` tag so that it appears similar to the following:

```
<c1chart:C1Chart Name="c1Chart1" Theme="{DynamicResource
{ComponentResourceKey TypeInTargetAssembly=c1chart:C1Chart,
ResourceId=Office2003Royale}}">
```

In Code

To specifically define the **Office2003Royale** theme in your chart add the following code your project:

- Visual Basic

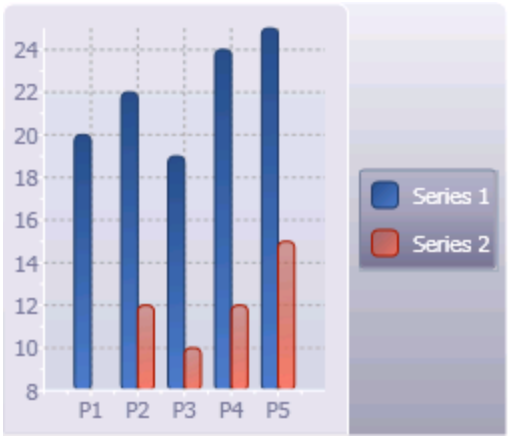
```
C1Chart1.Theme = TryCast(C1Chart1.TryFindResource( _
   New ComponentResourceKey(GetType(C1Chart), "Office2003Royale")), _
   ResourceDictionary)
```

- C#

```
c1Chart1.Theme = c1Chart1.TryFindResource(
   new ComponentResourceKey(typeof(C1.WPF.C1Chart.C1Chart),
   " Office2003Royale")) as ResourceDictionary;
```

Office2003Silver Theme

This theme is based on the **Office 2003 Silver** style and it appears as a silver-colored chart with gray and orange highlighting.



In XAML

To specifically define the **Office2003Silver** theme in your chart, add the following `Theme` XAML to the `<c1chart:C1Chart>` tag so that it appears similar to the following:

```
<c1chart:C1Chart Name="c1Chart1" Theme="{DynamicResource
{ComponentResourceKey TypeInTargetAssembly=c1chart:C1Chart,
ResourceId=Office2003Silver}}">
```

In Code

To specifically define the **Office2003Silver** theme in your chart, add the following code your project:

- Visual Basic

```
C1Chart1.Theme = TryCast(C1Chart1.TryFindResource( _
   New ComponentResourceKey(GetType(C1Chart), "Office2003Silver")), _
   ResourceDictionary)
```

- C#

```
c1Chart1.Theme = c1Chart1.TryFindResource(
   new ComponentResourceKey(typeof(C1.WPF.C1Chart.C1Chart),
   "Office2003Silver")) as ResourceDictionary;
```

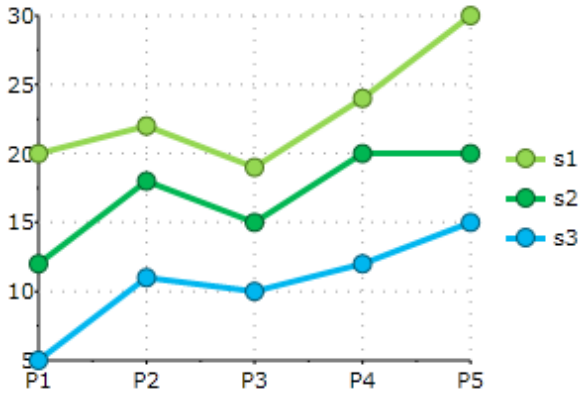
Data Series Color Generation

The data series color scheme can be selected by using the Palette property. By default, C1Chart uses the **ColorGeneration.Default** setting. The remaining options mimic the color themes of Microsoft Office.

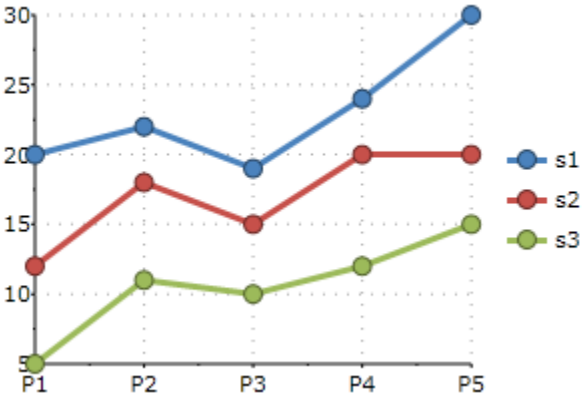
Available color schemes for the data series are listed below:

Color Generation Setting	Description or Preview
Default	When the **C1Chart.ColorGeneration** is set to "Default" the chart uses the theme palette if the theme is set, otherwise "Apex" palette is applied.

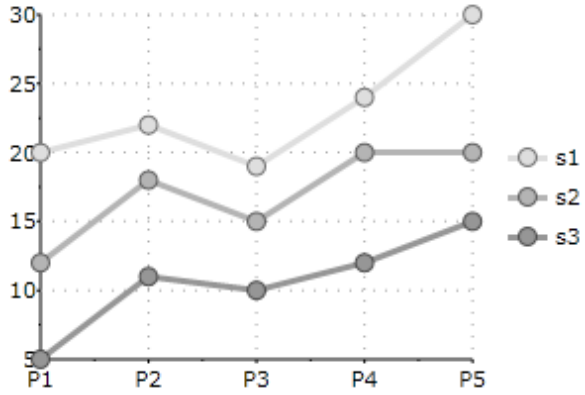
Standard



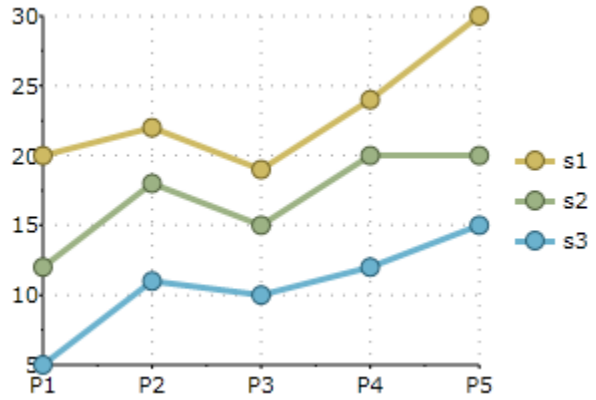
Office



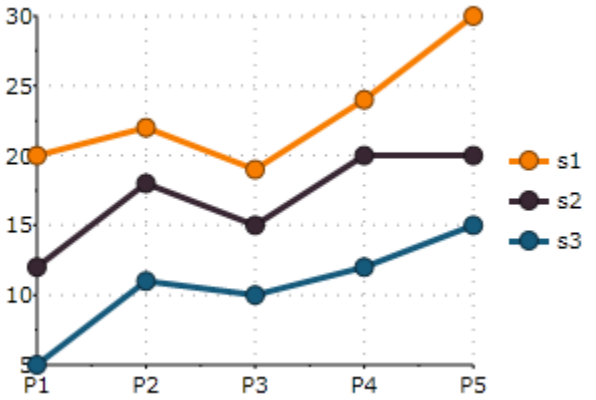
GrayScale



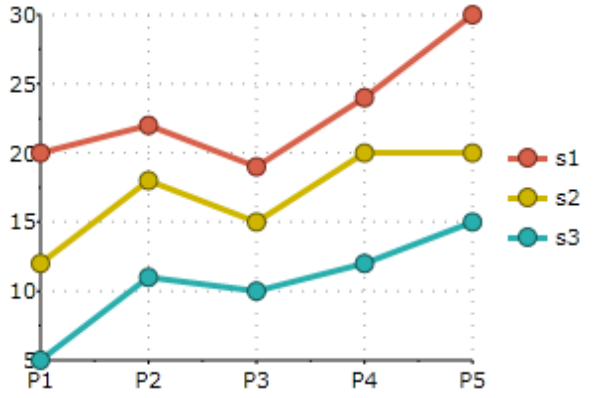
Apex



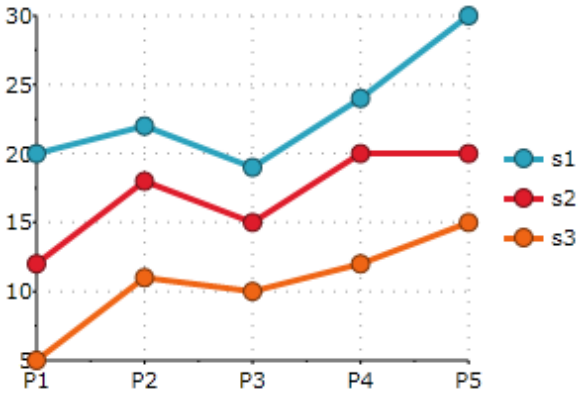
Aspect



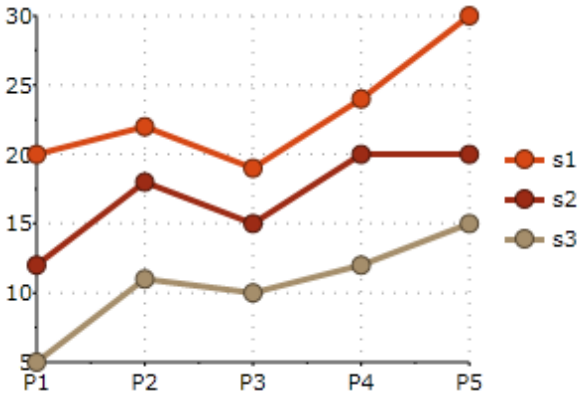
Civic



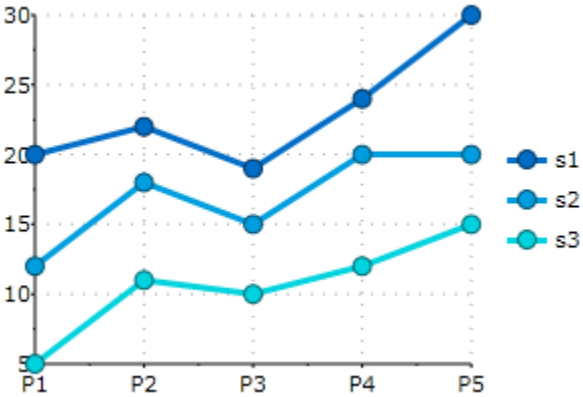
Concourse



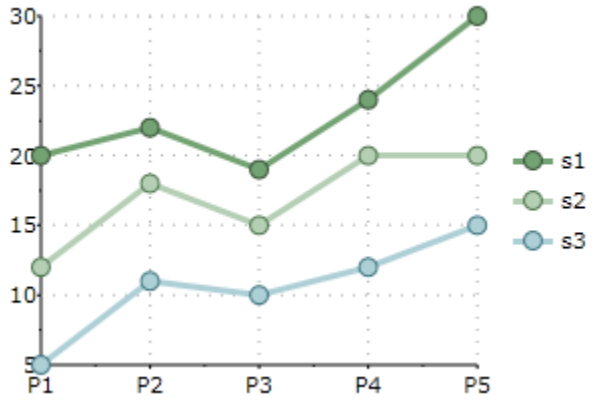
Equity



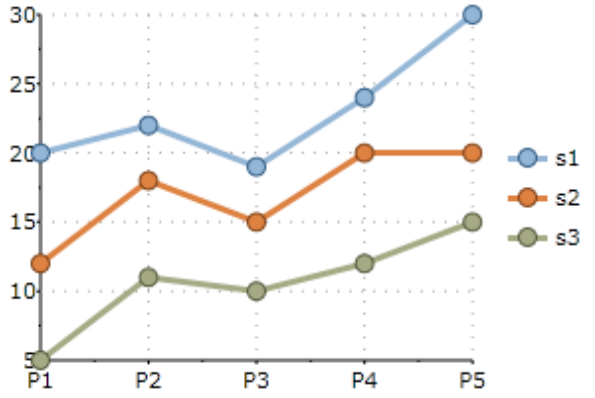
Flow



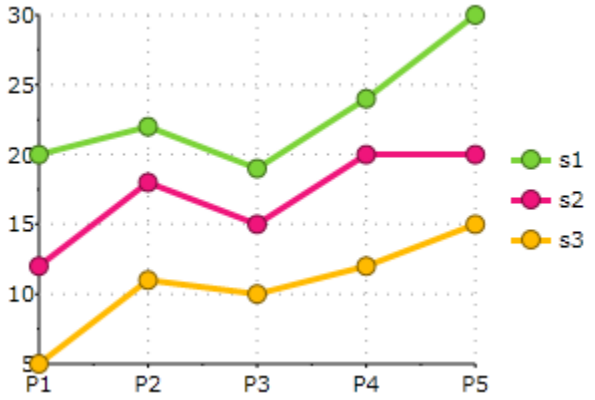
Foundry



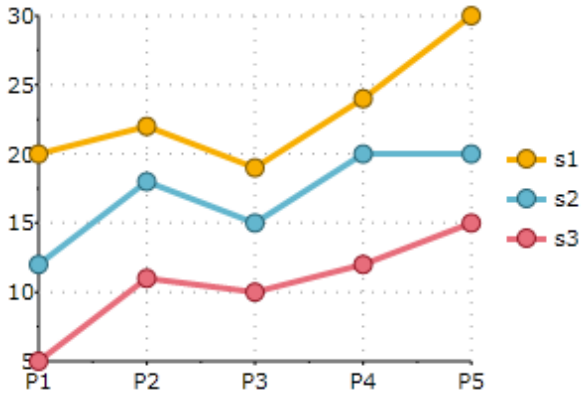
Median



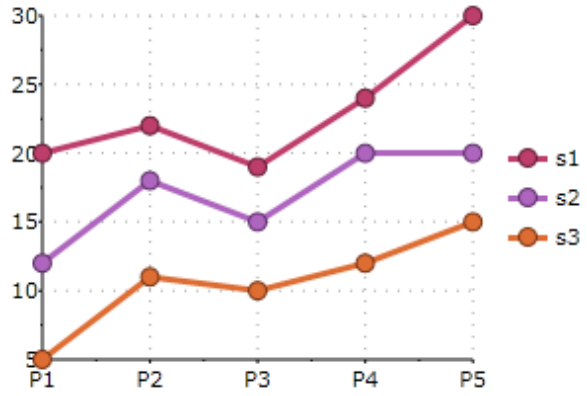
Metro



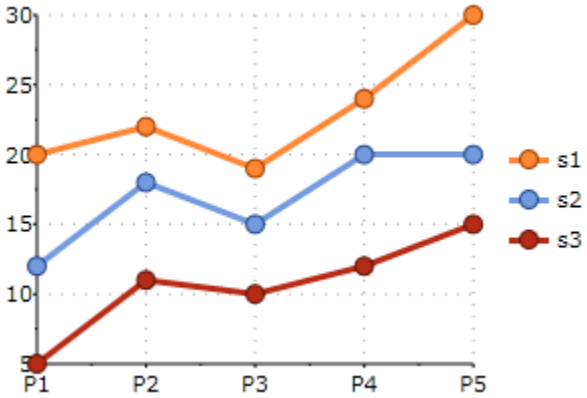
Module



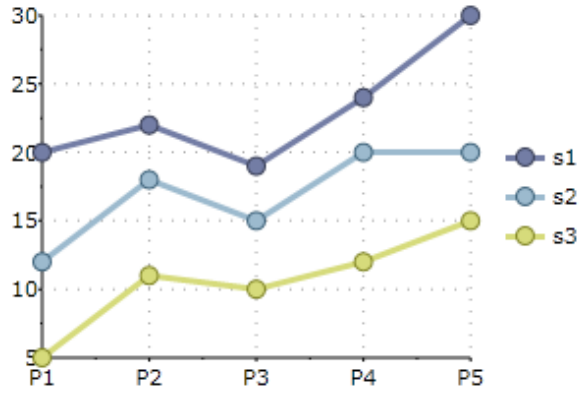
Opulent



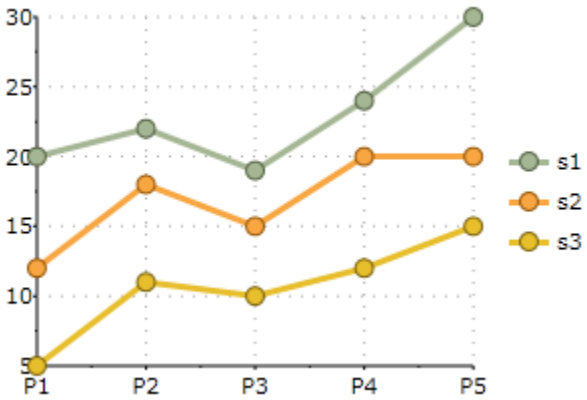
Oriel



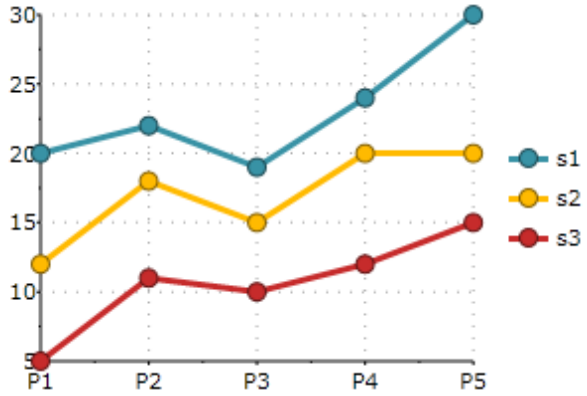
Origin



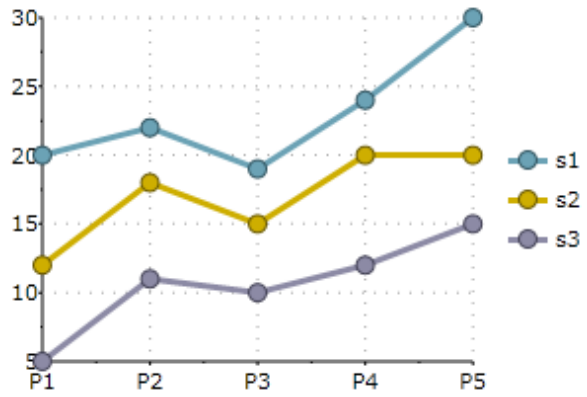
Paper



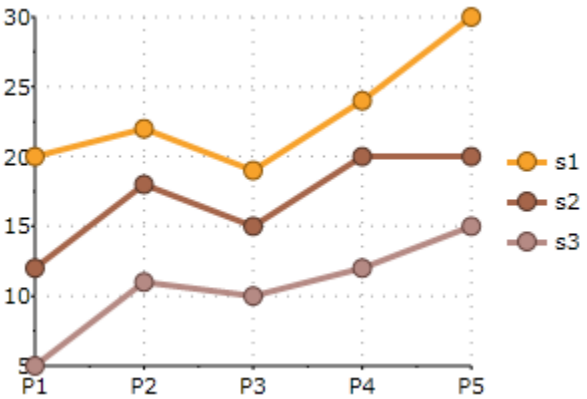
Solstice



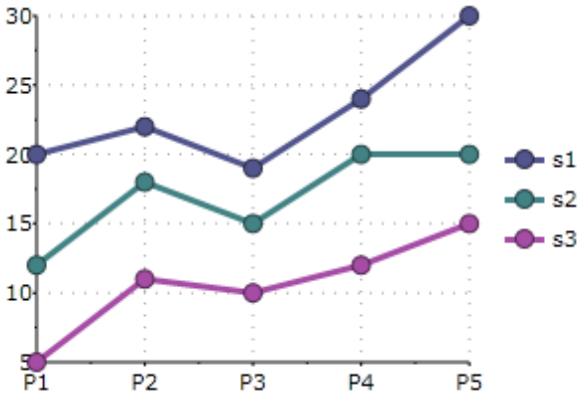
Technic



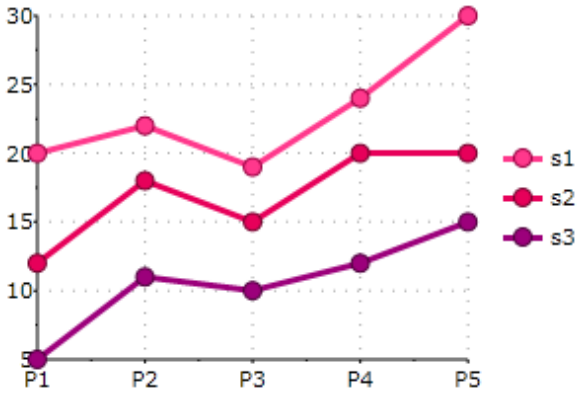
Trek



Urban



Verve



End User Interaction

C1Chart contains built-in tools that simplify the implementation of interactive behaviors for the end user. The end user can explore, rotate and zoom chart using combinations of mouse and shift keys. The control center for interactive features is the Actions property of C1Chart. The Action object has several properties that allow customization of the interface. All of the properties can be set or changed at design time through the Properties window with the **Action Collection Editor** or in XAML and programmatically through the Actions collection.

The following XAML shows how to set the Actions properties to enable end user interaction:

```
<c1:C1Chart.Actions>
                <c1:ZoomAction />
                <c1:TranslateAction Modifiers="Shift" />
                <c1:ScaleAction Modifiers="Control" />
            </c1:C1Chart.Actions>
```

The following code shows how to programatically set the Actions properties through the C1Chart.Actions collection:

```
c1.Chart.Actions.Add(new ZoomAction());
```

The following list reveals the supported chart actions:

- Rotate action allows changing viewing angle. This action is available only for chart with 3D effects. The Rotate3Daction class represents the rotate action for the 3D charts.
- Scale action increases or decreases the scale of the chart along the selected axis or axes. The ScaleAction class represents the scale action.

  Note: The zoom is not applicable for the chart's axis if the **MinScale** property is equal to zero. The **MinScale** property specifies the minimum scale that can be set for the axis.

- Translate action provides the opportunity to scroll through the plot area. The TranslateAction class represents the translate action.

  Note: You will not be available to translate along the axis if the Axis.Scale property is greater than 1.

- Zoom action allows the user to select rectangular area for viewing.

Note: The zoom is not applicable for the chart's axis if the **MinScale** property is equal to zero. The **MinScale** property specifies the minimum scale that can be set for the axis.

The scaling, translation and zooming are available only for chart with Cartesian axes.

Interactive rotation at run time is available for 3D Charts.

The Action object provides a set of properties that help to customize the action's behavior.

- The MouseButton and Modifiers properties specify the mouse button and key (ALT, CONTROL or SHIFT) combination that will invoke the execution of the action.

XAML Elements

Several auxiliary XAML elements are installed with **ComponentOne Chart for WPF**. These elements include templates and themes and are located in the Chart for WPF installation directory, by default **C:\Program Files\ComponentOne\Studio for WPF\C1WPFChart\XAML**. You can incorporate these elements into your project, for example, to create your own theme based on the included Office 2007 themes. For more information about the built-in themes some of these elements represent, see Chart Themes (page 119).

Included Auxiliary XAML Elements

The following auxiliary XAML elements are included with **Chart for WPF** with their location within the **C:\Program Files\ComponentOne\Studio for WPF\C1WPFChart\XAML** folder noted:

Element	Folder	Description
ChartTypes.xaml		Specifies the templates for all the available chart types.
default.xaml	Themes	Specifies the templates for Default theme.
DuskBlue.xaml	Themes	Specifies the templates for the Dusk Blue theme.
DuskGreen.xaml	Themes	Specifies the templates for the Dusk Green theme.
generic.xaml	Themes	Specifies the templates for different styles and the initial style of the chart.
Grayscale.xaml	Themes	Specifies the templates for the grayscale theme.
Legend.xaml	Themes	Specifies the templates for the Legend.
MediaPlayer.xaml	Themes	Specifies the templates for the Media Player theme.
Office2003Blue.xaml	Themes	Specifies the templates for the Office 2003 Blue theme.
Office2003Classic.xaml	Themes	Specifies the templates for the Office 2003 Classic theme.
Office2003Olive.xaml	Themes	Specifies the templates for the Office 2007 Olive theme.
Office2003Royale.xaml	Themes	Specifies the templates for the Office 2007 Royal theme.
Office2003Silver.xaml	Themes	Specifies the templates for the Office 2007 Silver theme.

Office2007Black.xaml	Themes	Specifies the templates for the Office 2007 Black theme.
Office2007Blue.xaml	Themes	Specifies the templates for the Office 2007 Blue theme.
Office2007Silver.xaml	Themes	Specifies the templates for the Office 2007 Silver theme.
Vista.xaml	Themes	Specifies the templates for the Vista theme.

Plotting Functions

C1Chart has a built-in engine for plotting functions. To use the built-in engine for plotting functions it is necessary to add a reference to the **C1.WPF.C1Chart.Extended.dll** in your project.

There are various types of functions used for different applications. **C1Chart** provides the various types of functions needed to create many applications.

There are two types of supported functions:

1. One-variable explicit functions
   - One-variable explicit functions are written as y=f(x) (see the YFunctionSeries class).
   - A few examples include: rational, linear, polynomial, quadratic, logarithmic, and exponential functions.
   - Commonly used by scientists and engineers, these functions can be used in many different types of finance, forecasts, performance measurement applications, and so on.
2. Parametric functions
   - The function is defined by a pair of equations, such as y=f1(t) and x=f2(t) where t is the variable/coordinate for functions f1 and f2.
   - Parametric functions are special types of function because the x and y coordinates are defined by individual functions of a separate variable.
   - They are used to represent various situations in mathematics and engineering, from heat transfer, fluid mechanics, electromagnetic theory, planetary motion and aspects of the theory of relativity, to name a few.
   - For more information about the parametric function (see the ParametricFunctionSeries class).

Using a Code String to Define a Function

When an interpretive code string is used to define a function of a function class (YFunctionSeries or ParametricFunctionSeries), the string is compiled and the resulting code is dynamically included into the application. Execution speed will be the same as any other compiled code.

For simple, one-variable explicit functions, the YFunctionSeries class object is used. This object has one code property, FunctionCode. For YFunction objects, the independent variable is always assumed to be "x".

For parametric functions, a pair of equations must be defined using the ParametricFunctionSeries class object. This object has two properties, one for each coordinate. The properties, XFunctionCode and YFunctionCode accept code in which the independent variable is always assumed to be "t".

Calculating the Value for Functions

You can calculate the value of the functions for Parametric and YFunction using the CalculateValue method.

TrendLines

The trend lines supported by chart with TrendLine objects can be divided into two groups, including regression and non-regression. In 2D charts, trend lines are typically used in X-Y line, bar, or scatter charts.

Non-regression trendlines are **MovingAverage**, **Average**, **Minimum**, and **Maximum**. Moving Average trendline is the average over the specified time.

Regression trend lines are **polynomial**, **exponent**, **logarithmic**, **power** and **Fourier functions** that approximate the data which the functions trend.

To use the trend lines feature, it is necessary to add the reference to the **C1.WPF.C1Chart.Extended.dll** in your project.

Chart Resource Keys

Built in themes and resources have several incorporated resource keys. These keys include brush, border, and other elements and can be customized to represent a unique appearance. When customizing a theme, a resource key that is not explicitly specified will revert to the default. The included resource keys and their descriptions are noted in the following topics.

The following tables describe chart resource keys for the chart control and its elements such as the chart area, plot area, axes, and legend.

Chart Resource Key

Resource Key	Description
C1Chart_Foreground_Color	Represents C1Chart's foreground color.
C1Chart_Background_Color	Represents C1Chart's background color.
C1Chart_Background_Brush	Represents C1Chart's background brush.
C1Chart_Foreground_Brush	Represents C1Chart's foreground brush.
C1Chart_Border_Brush	Represents C1Chart's border brush.
C1Chart_Border_Thickness	Represents C1Chart's border thickness (all 4 edges).
C1Chart_CornerRadius	Represents chart's corner radius (all 4 corners).
C1Chart_Padding	Represents C1Chart's padding.
C1Chart_Margin	Represents C1Chart's margin.

Legend Resource Keys

Resource Key	Description
C1Chart_LegendBackground_Brush	Represents C1Chart's legend background brush.
C1Chart_LegendForeground_Brush	Represents Legend's foreground brush for the C1Chart control.
C1Chart_LegendBorder_Brush	Represents Legend's border brush for the C1Chart control.

C1Chart_LegendBorder_Thickness	Represents the Legend border's thickness (all 4 edges) for the C1Chart control.
C1Chart_Legend_CornerRadius	Represents the Legend's corner radius (all 4 corners).

Chart Area Resource Keys

Resource Key	Description
C1Chart_ChartAreaBackground_Brush	Represents the ChartArea's background brush.
C1Chart_ChartAreaForeground_Brush	Represents the ChartArea's foreground brush on mouseover.
C1Chart_ChartAreaBorder_Brush	Represents the ChartArea's border brush.
C1Chart_ChartAreaBorder_Thickness	Represents the ChartArea's border thickness.
C1Chart_ChartArea_CornerRadius	Represents the ChartArea's corner radius (all 4 corners).
C1Chart_ChartArea_Padding	Represents the ChartAreas' padding.

Plot Area Resource Keys

Resource Key	Description
C1Chart_PlotAreaBackground_Brush	Represents the PlotArea's background brush.

Custom palette for plot elements Key

Resource Key	Description
C1Chart_CustomPalette	Represents the custom palette for plot elements.

Axis Keys

Resource Key	Description
C1Chart_AxisMajorGridStroke_Brush	Represents the AxisMajorGridStroke's brush.
C1Chart_AxisMinorGridStroke_Brush	Represents the AxisMinorGridStroke's brush.

Animation

Almost all of the plot elements can be animated with standard WPF animations. The following style is the modified style that adds "running ants" animation to the element which is under the mouse pointer.

```
<Style x:Key="mouseOver" TargetType="{x:Type c1c:PlotElement}">
   <!-- Default black outline -->
   <Setter Property="Stroke" Value="Black" />
   <Style.Triggers>
   <!-- When mouse is over element make thick red outline  -->
         <Trigger Property="IsMouseOver" Value="true">
               <Setter Property="Stroke" Value="Red" />
               <Setter Property="StrokeThickness" Value="2" />
               <Setter Property="StrokeDashArray" Value="2,2" />
               <Setter Property="Canvas.ZIndex" Value="1" />
               <Trigger.EnterActions>
                     <!-- Start animation -->
                     <BeginStoryboard >
                           <Storyboard>
```

```
                            <DoubleAnimation
Storyboard.TargetProperty="StrokeDashOffset"
                            From="0" To="8" RepeatBehavior="Forever"
Duration="0:0:0.5"/>
                            </Storyboard>
                        </BeginStoryboard>
                </Trigger.EnterActions>
            </Trigger>
    </Style.Triggers>
</Style>
```

Each DataSeries in a chart is composed of PlotElement objects that represent each individual symbol, connector, area, pie slice, etc in the series. The specific type of PlotElement depends on the chart type.

You can add animations to your charts by attaching **Storyboard** objects to the plot elements. This is usually done in response to the **DataSeries.Loaded** event, which fires after the PlotElement objects have been created and added to the data series.

Delivering Data to the Chart

Chart for WPF's **C1Chart** control can be bound to any object that implements the System.Collections.IEnumerable interface (such as XmlDataProvider, DataSet, DataView, and so on).

A datatable can be bound to the chart by assigning the **ItemsSource** property to the C1Chart control.

The following topics provide information on the different data binding methods used to deliver data to the C1Chart control.

Collection of Values

You can use several methods for delivering data to the chart. One method, is binding a collection of values using the ValuesSource property.

Any collection of numerical values that support **IEnumerable** interface can be used as a data source for the data series. Each data series class has appropriate properties for data binding. For example, the DataSeries class uses the ValuesSource property for data binding.

To bind the collection of values to the **DataSeries** you can first specify the binding source as an array of double like the following:

```
<!—Binding Source -->
<x:Array xmlns:sys="clr-namespace:System;assembly=mscorlib"
   x:Key="array" Type="sys:Double">
   <sys:Double>1</sys:Double>
   <sys:Double>4</sys:Double>
   <sys:Double>9</sys:Double>
   <sys:Double>16</sys:Double>
</x:Array>
```

To pass the array to the data series use the following code:

```
<!—Binding Target -->
<c1chart:DataSeries ValuesSource="{Binding Source={StaticResource
array},Path=Items}"/>
```

It is possible to specify the data values as an attribute, the values should be separated with spaces, for example:

```
<c1chart:DataSeries Values="1 2 9 16"/>
```

The preceding markup declaratively binds the ValuesSource property of a DataSeries to the *Items* property of a DataSeries object which is given a value of "1 2 9 16".

Collection of Objects

Data binding should be used when you have a collection of objects where each object includes numerical properties. There are at least two chart properties involved in the data binding process.

- ItemsSource property – The source where the collection of objects are assigned to.
- ValueBinding property – Gets or sets the value binding for the chart's data series. Specifies what object property provides the data value.

Suppose we have the array of points in resources:

```
<x:Array x:Key="points" Type="Point">
   <Point>0,0</Point>
   <Point>10,0</Point>
   <Point>10,10</Point>
   <Point>0,10</Point>
   <Point>5,5</Point>
</x:Array>
```

The following XAML fragment presents the chart with two data series, one is bound to X coordinate of points, and the other is bound to Y coordinate:

```
<c1chart:C1Chart Name="chart2">
   <c1chart:C1Chart.Data>
         <c1chart:ChartData
 ItemsSource="{Binding Source={StaticResource points}, Path=Items}">
              <c1chart:DataSeries ValueBinding="{Binding Path=X}"/>
              <c1chart:DataSeries ValueBinding="{Binding Path=Y}"/>
         </c1chart:ChartData>
   </c1chart:C1Chart.Data>
</c1chart:C1Chart>
```

The next sample shows the series that uses both coordinates of points at once; note it is the instance of XYDataSeries class that handles two sets of data values which correspond to x- and y-coordinates:

```
<c1chart:XYDataSeries
   XValueBinding="{Binding Path=X}"
   ValueBinding="{Binding Path=Y}"/>
```

Observable Collection

WPF has a special generic collection class **ObservableCollection** which provides notification about updating such as when items get added, removed, or when the entire list is refreshed. If an instance of this class is used as a data source for the chart, the chart automatically reflects the changes that were made in the collection.

Data Context Binding

If you plan to have multiple properties bind to the same source then use the **DataContext** property. The **DataContext** property provides a convenient way to establish a data scope.

The **C1Chart** control uses the **DataContext** property as ItemsSource when the items source is not set. **DataContext** should be IEnumerable for using as items source.

The following topics show how to use the **DataContext** as a Double and a Point:

Data Context as Array of Double

The following code shows how to use the data context as an array of double:

```
c1Chart1.Reset(true);
c1Chart1.DataContext = new double[] { 1, 2, 3, 4, 5 };
c1Chart1.ChartType = ChartType.Column;
```

Data Context as Array of Point

The following code shows how to use the data context as an array of point:

```
c1Chart1.Reset(true);
c1Chart1.DataContext = new Point[] { new Point(1, 1), new Point(2, 2),
new Point(3, 4), new Point(4, 1) };
c1Chart1.ChartType = ChartType.LineSymbols;
```

Data Series Binding

C1Chart provides the following binding types used for specifying which properties should be plotted on the chart:

- Item name binding - Specifies the item name binding for the chart data.
- Series bindings - Collection of value bindings for data series for each binding in the collection the data series is created during auto-generation.
- X-value binding- The X-Value binding specifies the x-value binding for the chart data series.

Item Name Binding

The Item Name binding is a type of data binding used to specify the item name binding for the chart data when the ItemNameBinding property is used.

The following example calls the bindings method on the target object:

```
ChartBindings bindings = new ChartBindings();
bindings.ItemNameBinding = new Binding("Name");
bindings.SeriesBindings.Add(new Binding("Input"));
bindings.SeriesBindings.Add(new Binding("Output"));

chart.Bindings = bindings;
chart.DataContext = new InOut[]
{
    new InOut() { Name = "n1", Input = 90, Output = 110},
    new InOut() { Name = "n2", Input = 80, Output = 70},
    new InOut() { Name = "n3", Input = 100, Output = 100},
};

where InOut is defined as:

public class InOut
{
    public string Name { get; set; }
    public double Input{ get; set;}
    public double Output { get; set; }
}
```

X-Value Binding

The X-Value binding specifies the x-value binding for the chart data series when the XBinding property is used.

The following example uses the XBinding property to set the x-value binding for the data series:

```
ChartBindings bindings = new ChartBindings();
bindings.XBinding = new Binding("X");
bindings.SeriesBindings.Add(new Binding("Y"));

chart.Bindings = bindings;
chart.DataContext = new Point[] { new Point(1, 0),
new Point(2, 2), new Point(3, 1), new Point(5, 3) };
```

Series Generation

The chart data series can be generated manually or automatically.

The property AutoGenerateSeries specifies whether the series are created automatically. By default the AutoGenerateSeries property is null and only generates data series if Children collection is empty. While the series are being generated, chart analyzes the elements of Data.ItemsSource (or C1Chart.DataContext) collection and creates series for each property that has supported types (numeric,DateTime).

To control the process of series generation, use the Bindings property that allows to specifiy which properties should be plotted. For more information on the Bindings property see, Data Series Binding (page 143).

Data Binding Tutorials

The following sections include data binding tutorials for the **C1Chart** control. The tutorials provide step-by-step instructions. By following the steps outlined in this chapter, you will be able to bind **C1Chart** to a data table and an xml file.

The key properties in both tutorials are the following:

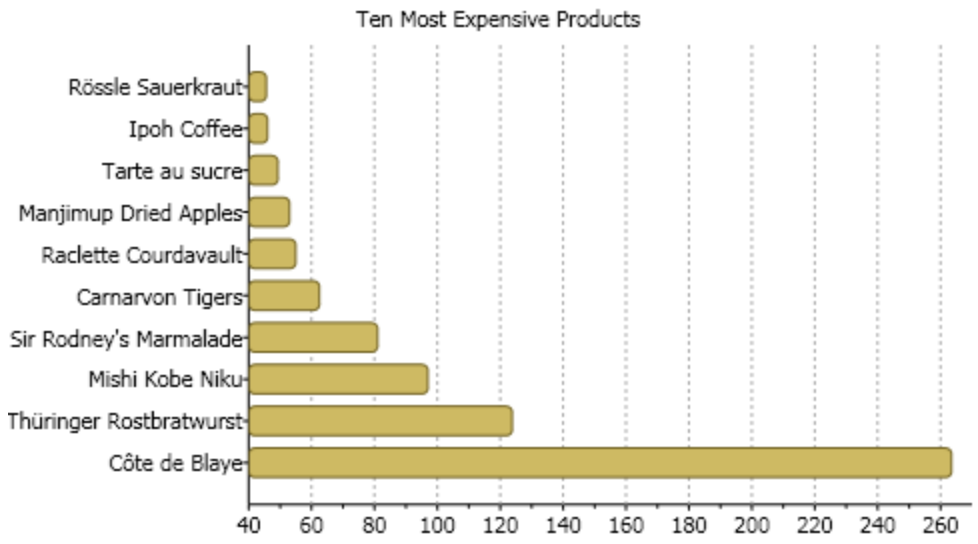
- ItemsSource - provides a list of objects
- ItemNameBinding - specifies the name of an element such as label on the x-axis
- ValueBinding - specifies the numerical value

Bind to a Data Table Declaratively

This tutorial provides step-by-step instructions for binding the **C1Chart** to a dataset declaratively. The data shows the information as a simple bar chart with one y-axis that represents the names of the products and one x-axis that represents the unit's price for each product. The products' ten most expensive products are displayed in descending order. The Bar chart uses one series to draw the unit price. A legend is not used since there is only one series.

The chart uses data from the sample Access database, Nwind.mdb. This tutorial assumes that the database file Nwind.mdb is in the C:**Users\username\Documents\ComponentOne Samples\Common (XP)** or **C:\Documents and Settings\username\My Documents\ComponentOne Samples\Common (Vista)** directory, and refer to it by filename instead of the full path name for the sake of brevity.

Completing this tutorial will produce a chart that looks like the following:



To bind C1Chart to a data table declaratively, complete the following steps:

1. Create a new WPF project in Visual Studio. For more information about creating a WPF project, see Creating a .NET Project in Visual Studio (page 23).
2. Add the **C1.WPF.C1Chart** reference to your project.
3. Add the **C1Chart** control to the Window. For more information see, Adding the Chart for WPF Components to a Visual Studio Project (page 25).
4. Once the **C1Chart** control is placed on the Window, the following XAML code is added:

```
Title="Window1" Height="300" Width="500" xmlns:c1chart="clr-
namespace:C1.WPF.C1Chart;assembly=C1.WPF.C1Chart" Loaded="Window_Loaded">
    <Grid>
        <c1chart:C1Chart Content="" Margin="10,10,10,18" Name="c1Chart1">
            <c1chart:C1Chart.Data>
                <c1chart:ChartData>
                    <c1chart:ChartData.ItemNames>P1 P2 P3 P4
P5</c1chart:ChartData.ItemNames>
                    <c1chart:DataSeries Label="Series 1" Values="20 22 19
24 25" />
                    <c1chart:DataSeries Label="Series 2" Values="8 12 10
12 15" />
                </c1chart:ChartData>
            </c1chart:C1Chart.Data>
            <c1chart:Legend DockPanel.Dock="Right" />
        </c1chart:C1Chart>
    </Grid>
```

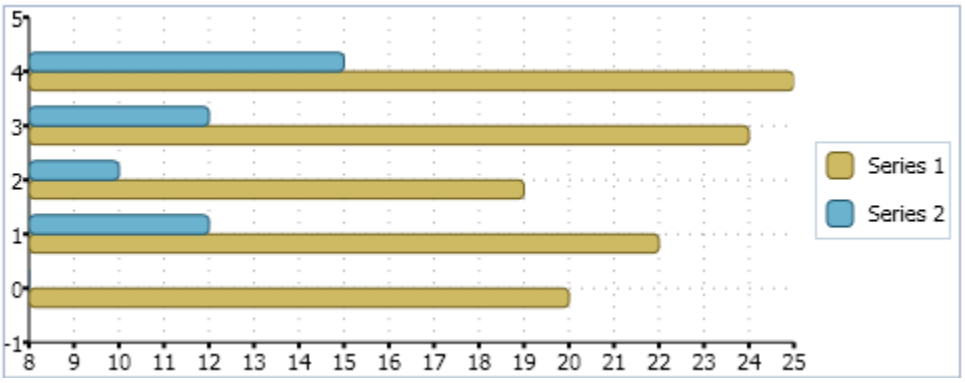
5. Select the XAML tab and add the following namespace in the XAML code:

```
  xmlns:sys="clr-namespace:System;assembly=mscorlib"
```

6. In the XAML code, change the Title's **Width** from 300 to 500.
7. Within the `<c1chart:C1Chart>` tag modify the **Margin** to "0" and set the **ChartType** to "Bar". This will change the default chart's appearance from **Column** to **Bar**. Your XAML code should appear like the following:

```
<c1chart:C1Chart Margin="0" Name="c1Chart1" ChartType="Bar">
</c1chart:C1Chart>
```

   Your chart appears like the following:



8. Create a label after the closing c1chart:C1Chart tag and label it as "Ten Most Expensive Products".

```
<TextBlock DockPanel.Dock="Top" Text="Ten Most Expensive Products"
HorizontalAlignment="Center"/>
```

Your XAML code should now appear like the following:

```
<Grid>
        <c1chart:C1Chart Margin="0" Name="c1Chart1" ChartType="Bar"
Height="185" VerticalAlignment="Top">
            <c1chart:C1Chart.Data>
                <c1chart:ChartData>
                    <c1chart:DataSeries Label="Series 1" Values="20 22 19
24 25" />
                    <c1chart:DataSeries Label="Series 2" Values="8 12 10
12 15" />
                </c1chart:ChartData>
            </c1chart:C1Chart.Data>
            <c1chart:Legend DockPanel.Dock="Right" />
        </c1chart:C1Chart>
        <TextBlock DockPanel.Dock="Top" Text="Ten Most Expensive Products"
HorizontalAlignment="Center"/>
</Grid>
```

9. Add the following using/imports directives in your code-behind file:

- Visual Basic

```
Imports System.Data
Imports System.Data.OleDb
Imports C1.WPF.C1Chart
```

- C#

```
using System.Data;
using System.Data.OleDb;
using C1.WPF.C1Chart;
```

10. Declare the variable for the DataSet outside the **Window_Loaded** procedure, then add the following code to retrieve the products and unit price from the database:

- Visual Basic

```
Private _dataSet As DataSet
```

```
Private Sub Window_Loaded(ByVal sender As Object, ByVal e As
RoutedEventArgs)

    ' create connection and fill data set
    Dim mdbFile As String = "c:\Program Files\ComponentOne Studio.NET
2.0\Common\nwind.mdb"
    Dim connString As String =
String.Format("Provider=Microsoft.Jet.OLEDB.4.0; Data Source={0}",
mdbFile)
    Dim conn As New OleDbConnection(connString)
    Dim adapter As New OleDbDataAdapter("SELECT TOP 10 ProductName,
UnitPrice" & Chr(13) & "" & Chr(10) & " FROM Products ORDER BY
UnitPrice DESC;", conn)

    _dataSet = New DataSet()
    adapter.Fill(_dataSet, "Products")

    ' set source for chart data
    c1Chart1.Data.ItemsSource = _dataSet.Tables("Products").Rows
End Sub
```

- C#

```
DataSet _dataSet;

    private void Window_Loaded(object sender, RoutedEventArgs e)
    {

      // create connection and fill data set
        string mdbFile = @"c:\Program Files\ComponentOne Studio.NET
2.0\Common\nwind.mdb";
      string connString =
string.Format("Provider=Microsoft.Jet.OLEDB.4.0; Data Source={0}",
mdbFile);
      OleDbConnection conn = new OleDbConnection(connString);
  OleDbDataAdapter adapter =
    new OleDbDataAdapter(@"SELECT TOP 10 ProductName, UnitPrice
    FROM Products ORDER BY UnitPrice DESC;", conn);
      _dataSet = new DataSet();
      adapter.Fill(_dataSet, "Products");

      // set source for chart data
      c1Chart1.Data.ItemsSource = _dataSet.Tables["Products"].Rows;
    }
```

Note: Make sure the file path for the mdbFile matches up to where you have the nwind.mdb database project located on your machine.

11. Click on the XAML tab so your are in XAML view and delete the following default data from ChartData:

```
<c1chart:ChartData.ItemNames>P1 P2 P3 P4 P5</c1chart:ChartData.ItemNames>
                    <c1chart:DataSeries Label="Series 1" Values="20 22 19
24 25" />
                    <c1chart:DataSeries Label="Series 2" Values="8 12 10
12 15" />
```

The C1Chart control now appears empty on the Window.

12. Within the `<c1chart:C1Chart.Data>` tag add the **ItemNameBinding** property to the **ChartData** to specify the name of the element, in this case the label on the y-axis and the **ValueBinding** property to the **DataSeries** to specify the numerical value for the series.

```
<c1chart:ChartData ItemNameBinding="{Binding Path=[ProductName]}">
                    <c1chart:DataSeries ValueBinding="{Binding
Path=[UnitPrice]}"/>
                </c1chart:ChartData>
```

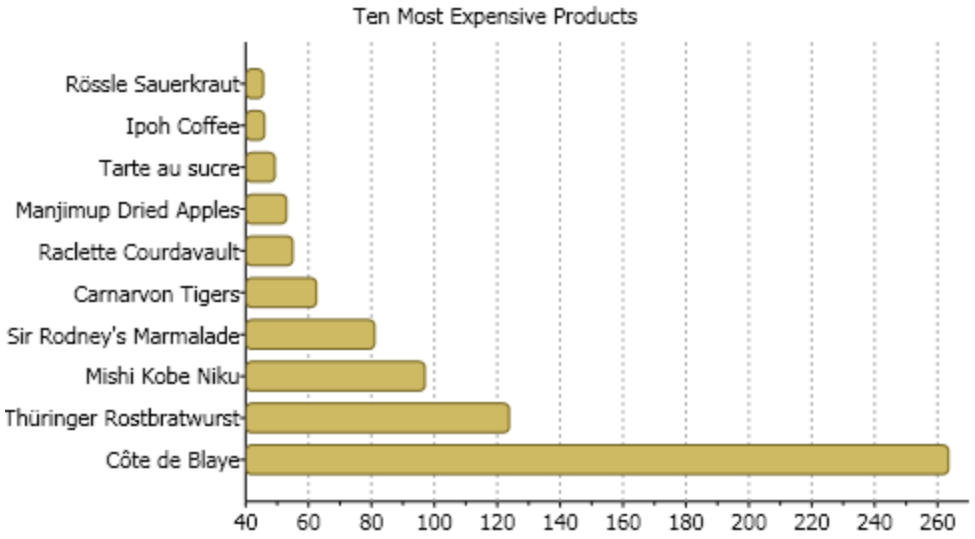
Your XAML code for your project should look like the following:

```
<Window x:Class="Chart for WPF_QuickStart.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:sys="clr-namespace:System;assembly=mscorlib"

    Title="Window1" Height="300" Width="500" Loaded="Window_Loaded"
xmlns:c1chart="clr-namespace:C1.WPF.C1Chart;assembly=C1.WPF.C1Chart">
    <Grid>
        <c1chart:C1Chart Margin="0" Name="c1Chart1" ChartType="Bar">
            <TextBlock DockPanel.Dock="Top" Text="Ten Most Expensive
Products"
                 HorizontalAlignment="Center"/>
            <c1chart:C1Chart.Data>
                <c1chart:ChartData ItemNameBinding="{Binding
Path=[ProductName]}">
                    <c1chart:DataSeries ValueBinding="{Binding
Path=[UnitPrice]}"/>
                </c1chart:ChartData>
            </c1chart:C1Chart.Data>
        </c1chart:C1Chart>
    </Grid>
</Window>
```

13. Remove the `<c1chart:Legend DockPanel.Dock="Right" />` tag from XAML to remove the built-in Legend control.
14. Run your project to ensure that everything is working correctly.

Observe the following at runtime

The chart is now populated with data from the Products table.



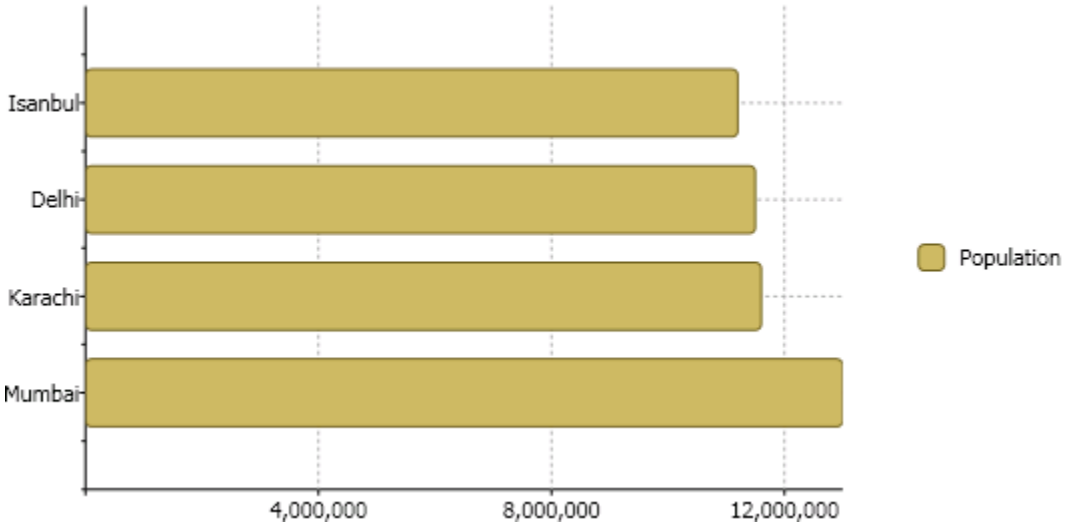
Bind to an XML

This tutorial provides step-by-step instructions for embedding the XML as a data island within the source of the XAML page to bind the **C1Chart** control to the xml data. The data shows the information as a simple bar chart with one y-axis that represents the names of the cities and one x-axis that represents the population for each country. The **Bar** chart uses one series to draw the population. A legend is used to show the color for the population.

In this tutorial the binding is set in the ChartData class using the following XAML code:

```
<c1chart:ChartData ItemsSource="{Binding Source={StaticResource data}}"
   ItemNameBinding="{Binding XPath=CityName}">
     <c1chart:DataSeries Label="Population"
       ValueBinding="{Binding XPath=Population}" />
</c1chart:ChartData>
```

Completing this tutorial will produce a chart that looks like the following:



To bind C1Chart to xml:

1. Create a new WPF project in Visual Studio. For more information about creating a WPF project, see Creating a .NET Project in Visual Studio (page 23).
2. Create a Resource section in your Window and add an XML data provider to it. Within the resources section embed the XML data directly as an XML data island. An XML data island must be wrapped in <x:Xdata> tags and always have a single root node, which is Cities in this example:

   XAML:

```
<Grid.Resources>
      <XmlDataProvider x:Key="data" XPath="Cities/City">
        <x:XData>
          <Cities xmlns="">
            <City>
              <CityName>Mumbai</CityName>
              <Population>13000000</Population>
            </City>
            <City>
              <CityName>Karachi</CityName>
              <Population>11600000</Population>
            </City>
            <City>
              <CityName>Delhi</CityName>
              <Population>11500000</Population>
            </City>
            <City>
              <CityName>Isanbul</CityName>
              <Population>11200000</Population>
            </City>
          </Cities>
        </x:XData>
      </XmlDataProvider>
</Grid.Resources>
```

3. Add the **C1.WPF.C1Chart** reference to your project.
4. Add the **C1Chart** control to the Window.

   Once the **C1Chart** control is placed on the Window, the following XAML code is added:

```
Title="Window1" Height="50" Width="100" xmlns:c1chart="clr-
namespace:C1.WPF.C1Chart;assembly=C1.WPF.C1Chart" Loaded="Window_Loaded">
    <Grid>
        <c1chart:C1Chart Content="" Margin="10,10,10,18" Name="c1Chart1">
            <c1chart:C1Chart.Data>
                <c1chart:ChartData>
                    <c1chart:ChartData.ItemNames>P1 P2 P3 P4
P5</c1chart:ChartData.ItemNames>
                    <c1chart:DataSeries Label="Series 1" Values="20 22 19
24 25" />
                    <c1chart:DataSeries Label="Series 2" Values="8 12 10
12 15" />
                </c1chart:ChartData>
            </c1chart:C1Chart.Data>
            <c1chart:Legend DockPanel.Dock="Right" />
        </c1chart:C1Chart>
    </Grid>
```

5. Change the Window's **Width** to "300" and **Height** to "550"

6. Within the <c1chart:C1Chart> tag modify the **Margin** to "0" and set the **ChartType** to "Bar". This will change the default chart's appearance from Column to Bar. Your XAML code should appear like the following:

```
<c1chart:C1Chart Margin="0" Name="c1Chart1" ChartType="Bar">

</c1chart:C1Chart>
```

7. In the XAML file locate the <c1chart:C1Chart.Data> tag and delete the following XAML code from it:

```
<c1chart:ChartData.ItemNames>P1 P2 P3 P4 P5</c1chart:ChartData.ItemNames>
                    <c1chart:DataSeries Label="Series 1" Values="20 22 19
24 25" />
                    <c1chart:DataSeries Label="Series 2" Values="8 12 10
12 15" />
```

   The two default series are removed from **C1Chart** and now the **C1Chart** control appears empty because there is no data for it.

8. Within the <c1chart:C1Chart.Data> tag add the **ItemNameBinding** property to the **ChartData** to specify the name of the element, in this case the label on the y-axis and the **ValueBinding** property to the DataSeries to specify the numerical value for the series. The following example binds the **ChartData.ItemsSource** property using the binding extension, specifying the Source. The **ChartData.ItemNameBinding** property is bound using the binding extension specifying the Path. The **DataSeries.Label** property is bound using the binding extension to specify the Path which is the Population.

   In XAML:

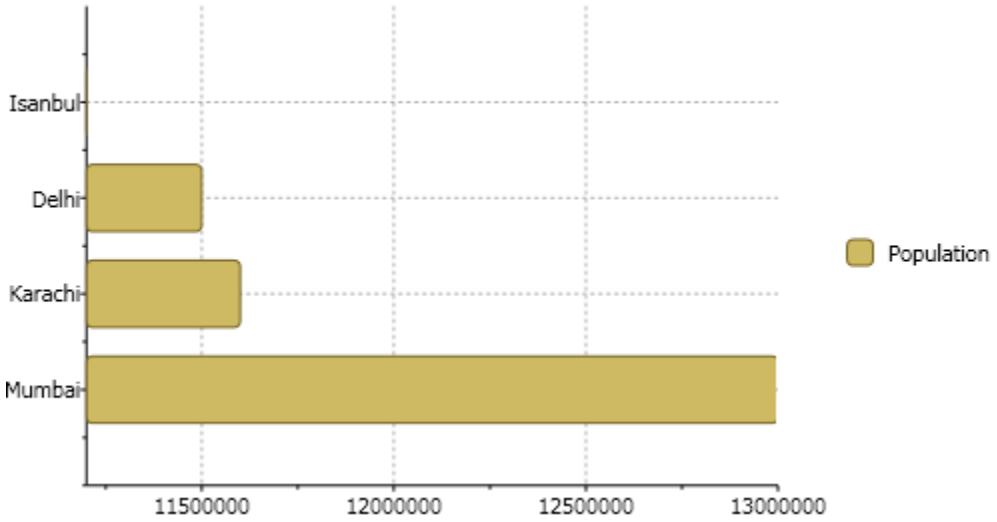
```
<c1chart:ChartData ItemsSource="{Binding Source={StaticResource data}}"
                     ItemNameBinding="{Binding XPath=CityName}">
          <c1chart:DataSeries Label="Population"
                       ValueBinding="{Binding XPath=Population}" />
        </c1chart:ChartData>
```

   Your XAML code for your C1Chart control should look like the following:

```
<c1chart:C1Chart Height="300"  HorizontalAlignment="Left" Margin="0"
Name="c1Chart1" ChartType="Bar" VerticalAlignment="Top" Width="500">
            <c1chart:C1Chart.Data>
                <c1chart:ChartData ItemsSource="{Binding
Source={StaticResource data}}"
                     ItemNameBinding="{Binding XPath=CityName}">
                    <c1chart:DataSeries Label="Population"
                         ValueBinding="{Binding XPath=Population}" />
                </c1chart:ChartData>
            </c1chart:C1Chart.Data>
            <c1chart:Legend DockPanel.Dock="Right" />
        </c1chart:C1Chart>
```

9. Run your project to ensure that everything is working correctly.

   Your chart will appear like the following:



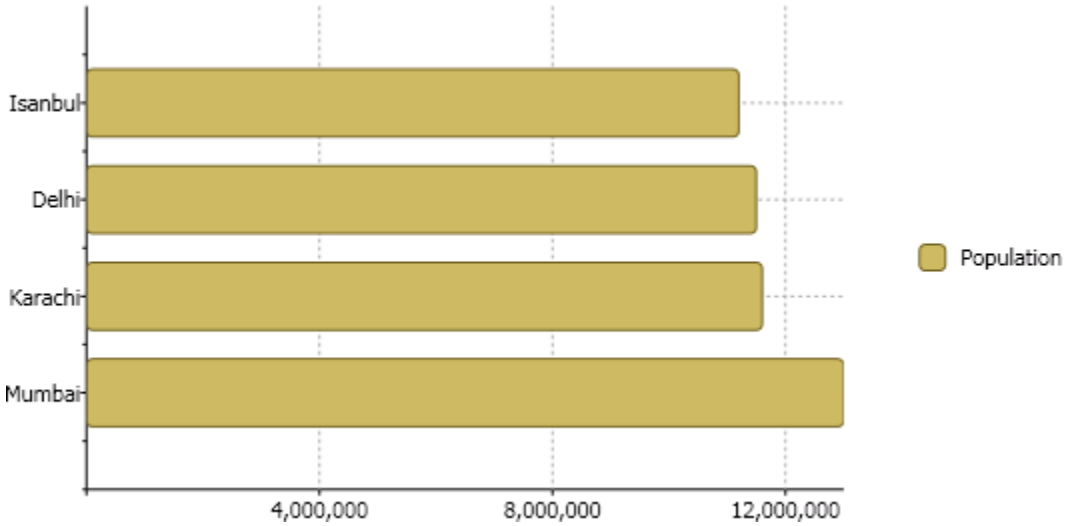
Notice how the annotation appears for the x-axis. We will need to format the annotation for the x-axis so the population appears in the thousandths.

10. Declare tags for C1Chart's **ChartView.AxisX** property. You will need to set the following properties of the **AxisX** to format the annotation and the gridlines.

    Add the following XAML code after the closing, `</c1chart:C1Chart.Data>`, tag:

```
<c1chart:C1Chart.View>
        <c1chart:ChartView>
          <c1chart:ChartView.AxisX >
            <c1chart:Axis Min="0" MajorGridStroke="DarkGray"
AnnoFormat="#,###,###"/>
          </c1chart:ChartView.AxisX>
        </c1chart:ChartView>
      </c1chart:C1Chart.View>
```

    The X-Axis annotation appears updated on the Chart like the following:



Using MVVM

ComponentOne **Chart for WPF** supports the MVVM (Model-View-ViewModel) design pattern. The entire chart can be declaratively written and bound to in XAML using native WPF binding techniques.

The following steps demonstrate how to use **C1Chart** in an MVVM-designed application.

Step 1: Creating the Model

Create a new class named **Sale**, which implements the **INotifyPropertyChanged** interface.

```
public class Sale : INotifyPropertyChanged
{
    private string _product;
    private double _value;
    private double _discount;

    public Sale(string product, double value, double discount)
    {
        Product = product;
        Value = value;
        Discount = discount;
    }

    public string Product
    {
        get { return _product; }
        set
        {
            if (_product != value)
            {
                _product = value;
                OnPropertyChanged("Product");
            }
        }
    }

    public double Value
    {
        get { return _value; }
        set
        {
            if (_value != value)
            {
                _value = value;
                OnPropertyChanged("Value");
            }
        }
    }

    public double Discount
    {
        get { return _discount; }
        set
        {
            if (_discount != value)
            {
```

```
                _discount = value;
                OnPropertyChanged("Discount");
            }
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;

    void OnPropertyChanged(string propertyName)
    {
        if (PropertyChanged != null)
            PropertyChanged(this, new
PropertyChangedEventArgs(propertyName));
    }
}
```

This class has several properties which define a Sale, including Product, Value and Discount.

By implementing **INotifyPropertyChanged** this enables binding properties to automatically reflect dynamic changes. For each property you want change notifications for, you call **OnPropertyChanged** whenever the property is updated. Note that **ObservableCollections** already inherit **INotifyPropertyChanged**.

Step 2: Creating the View Model

Create a new class named **SaleViewModel**. This will act as the DataContext for the View which will contain **C1Chart**.

public class SaleViewModel : INotifyPropertyChanged

```
{
    private ObservableCollection<Sale> _sales = new
ObservableCollection<Sale>();

    public SaleViewModel()
    {
        //load data
        LoadData();
    }

    public ObservableCollection<Sale> Sales
    {
        get { return _sales; }
    }

    public void LoadData()
    {
        //TODO: load data from your data source
        _sales.Add(new Sale("Bikes", 23812.89, 12479.44));
        _sales.Add(new Sale("Shirts", 79752.21, 19856.86));
        _sales.Add(new Sale("Helmets", 63792.05, 16402.94));
        _sales.Add(new Sale("Pads", 30027.79, 10495.43));
    }

    public event PropertyChangedEventHandler PropertyChanged;

    void OnPropertyChanged(string propertyName)
    {
        if (PropertyChanged != null)
```

```
            PropertyChanged(this, new
PropertyChangedEventArgs(propertyName));
    }
}
```

This class includes an **ObservableCollection**, **Sales**, as well as a method to generate mock data upon initialization.

Step 3: Creating the View Using C1Chart

1. Create a new WPF UserControl named SaleView.xaml and add the following XAML above the LayoutRoot grid:

```
<UserControl.Resources>
    <local:SaleViewModel x:Key="viewModel" />
</UserControl.Resources>
<UserControl.DataContext>
    <Binding Source="{StaticResource viewModel}"/>
</UserControl.DataContext>
```

   This XAML declares a SaleViewModel as a Resource and sets it to the DataContext of the UserControl. At runtime, the View will now be bound to the ViewModel. Controls within the View can now bind to public properties of the ViewModel.

2. Add a **C1Chart** control to the page.
3. Replace the XAML for **C1Chart** with the following code:

```
<c1:C1Chart ChartType="Column" Name="c1Chart1" Palette="Module">
    <c1:C1Chart.Data>
        <c1:ChartData ItemsSource="{Binding Sales}"
ItemNameBinding="{Binding Product}">
            <c1:DataSeries Label="Value" ValueBinding="{Binding Value}" />
            <c1:DataSeries Label="Discount" ValueBinding="{Binding
Discount}" />
        </c1:ChartData>
    </c1:C1Chart.Data>
    <c1:C1ChartLegend />
</c1:C1Chart>
```

   This XAML defines a **C1Chart** with two data series. The ChartData's **ItemsSource** is set to the collection of Sales objects exposed by our ViewModel. Each DataSeries has its ValueBinding property set and we also set the ItemNameBinding to display our product names along the X-axis.

   Note: If you are using **XYDataSeries**, then you should specify the XValueBinding for each series and you should not set the ItemNameBinding.

4. Open the **App.xaml.cs** application configuration file and in the **Application_Startup** event, replace the following code:

```
private void Application Startup(object sender, StartupEventArgs e)
{
    this.RootVisual = new Views.SaleView();
}
```

   This code sets the RootVisual to show your SaleView upon startup.

5. Run your application and observe that the **C1Chart** appears to be bound to the Sales data from the ViewModel.

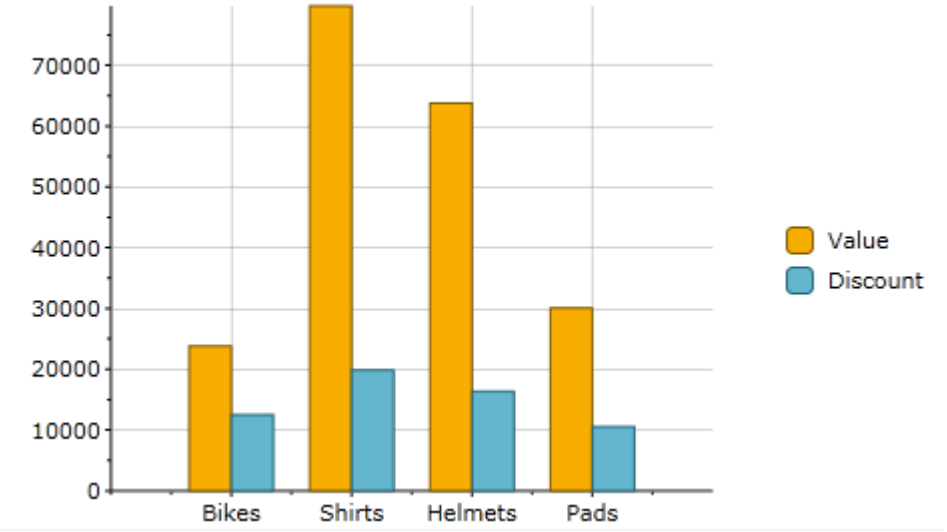


Chart for WPF Samples

Please be advised that this ComponentOne software tool is accompanied by various sample projects and/or demos, which may make use of other ComponentOne development tools included with ComponentOne Studios.

Samples can be accessed from the **ComponentOne Sample Explorer**. On your desktop, click the **Start** button and then click **ComponentOne | Studio for WPF | C1WPFChart Samples**. The following tables provide a short description for each sample included in the C1WPFChartSamples.

C1WPFChart Samples

Sample	Description
Advanced	The Advanced group includes the following samples: Aggregate, Functions, Moving Average, Parametric, Polygon, and Trend lines. The Aggregate sample, AggregateChart.xaml, shows the built-in aggregate features. The Functions sample, Functions.xaml, demonstrates how to plot the function represented by javascript expression. The Moving Average sample, MovAverage.xaml, creates two moving average trend lines with different periods. The Parametric sample, Parametric.xaml, plots the function defined parametrically with two equations x = x(t) and y = y(t). The Trend lines sample, TrendLines.xaml, shows available trend line types.
Appearance	The Appearance group includes two samples: Animation.xaml and Bubble.xaml. The Bubble sample shows how to create a complex Bubble chart and the Animation sample shows how to play animation when loading new data.
Axes	The Axes sample includes a Logarithmic, Ticks, Origin, Dependent Axes, and Custom Labels samples. The Logarithmic sample, LogAxes.xaml, shows logarithmic axes with default and custom axis templates. The Ticks sample, TickEditor.xaml, demonstrates various options for axis ticks. The Origin sample, AxisOrigin.xaml, shows how to use the scrollbars to change the position of origin for the x and y axes. The dependent axes sample,

	DependentAxes.xaml, demonstrates how to use dependent axes to display the temperature in different units. The Custom Labels sample, CustomLabels.xaml, shows how to customize the position and appearance of axis labels.
Basics	This sample includes the following samples: Labels and tooltips, Radar, Image, and export. The Lables and tooltips sample, Labels.xaml, shows how to attach labels or tooltips to each data point. The Radar sample, Radar.xaml, demonstrates how to use a Radar chart, and the Image and export sample, ImageExport.xaml, demonstrates how to export chart to an image file.
Combination	The Combination sample includes the following samples: Financial, Complex, and Plot areas. The Financial sample, FinancialChart.xaml, shows a financial chart with column and candle data series and two y-axes. The Complex sample, ComplexChart.xaml, shows a stacked column chart with two stacking groups and line data series.
Interaction	The Interaction group includes the following samples: Interactive zoom, AddRemove Markers, Markers, Mouse Marker, and Drag and Drop. The Interactive zoom sample, InteractiveChart.xaml, shows a zoom, scale and pan a chart using the mouse. The Drag and Drop sample, DragDrop.xaml, shows a drag and drop data series between the charts.
Performance	The Performance group includes the following samples: Live Data, Dynamic,xaml and Large data, LargeData.xaml. The Live data sample shows dynamic data with trend lines and the Large Data sample shows an interactive chart with 50000 data points.

To run this sample:

1. Open Visual Studio or Microsoft Expression Blend.
2. Select **File | Open Project/Solution**.
3. Click the drop-down Look in list and find the "<PersonalDocumentsFolder>\ComponentOne Samples\Common" where <PersonalDocumentsFolder> is the users Documents folder. This is the default location of the sample created by the installation program. The location may be different if you installed Chart for WPF elsewhere on your machine.
4. Select the Samples.sln and click **Open** to open the project in Blend or Visual Studio
5. Select **Project | Test Solution** or click **F5** to run the sample.

Chart for WPF Task-Based Help

The task-based help assumes that you are familiar with programming in Visual Studio .NET, and know how to use the C1Chart control in general. Each topic in this section provides a solution for specific tasks using the **ComponentOne Chart for WPF** product.

Each task-based help topic also assumes that you have created a new WPF project. For additional information on this topic, see Creating a .NET Project in Visual Studio (page 23) or Creating a Microsoft Blend Project (page 22).

Note: Some of the examples reference the C1NWind.mdb database which is installed by default in the **ComponentOne Samples\Common** folder installed in your **MyDocuments** folder (**Documents** in Vista).

Axes Tasks

The following topics show how to modify the axes origin, tick marks, and annotation.

Displaying Axis Labels on an Angle

You can display the labels for the X-Axis or Y-Axis on an angle using the AnnoAngle. The following code shows how to display the X-Axis annotation on a 90 degree angle:

```
chart.View.AxisX.AnnoAngle = -90;
```

Creating a Custom Annotation

To create a custom annotation using the AnnoTemplate property, use the following XAML or C# code:

XAML Code

```
...
    <c1chart:ChartView.AxisX>
      <c1chart:Axis>
        <c1chart:Axis.Resources >
          <local:ColorConverter x:Key="clrcnv" />
        </c1chart:Axis.Resources>
        <c1chart:Axis.AnnoTemplate>
          <DataTemplate>
            <TextBlock Width="25" TextAlignment="Center"
             Text="{Binding Path=Value}"
             Foreground="{Binding Converter={StaticResource clrcnv}}"/>
          </DataTemplate>
        </c1chart:Axis.AnnoTemplate>
      </c1chart:Axis>
    </c1chart:ChartView.AxisX>
...
```

C# Code

```
public class ColorConverter : IValueConverter {
   int cnt = 0;
   public object Convert(object value, Type targetType,
     object parameter, CultureInfo culture)
   {
     //DataPoint dpt = (DataPoint)value;
     // alternate the brushes
     return cnt++ % 2 == 0 ? Brushes.Blue : Brushes.Red;
```

```
    }

    public object ConvertBack(object value, Type targetType,
     object parameter, CultureInfo culture)
    {
      return null;
    }
}
```

Setting the Axis Origin

You can specify the axis origin using the Origin property, like the following:

```
{
          c1Chart1.Reset(true);
          c1Chart1.Data.Children.Add(new XYDataSeries()
                    {
                      ValuesSource = new double[] { -1, 2, 0, 2, -2 },
                      XValuesSource = new double[] { -2, -1, 0, 1, 2 }
                    };
          c1Chart1.View.AxisX.Origin = 0;
          c1Chart1.View.AxisY.Origin = 0;
          c1Chart1.ChartType = ChartType.LineSymbols;
        }));
```

Specifying the Major and Minor Ticks

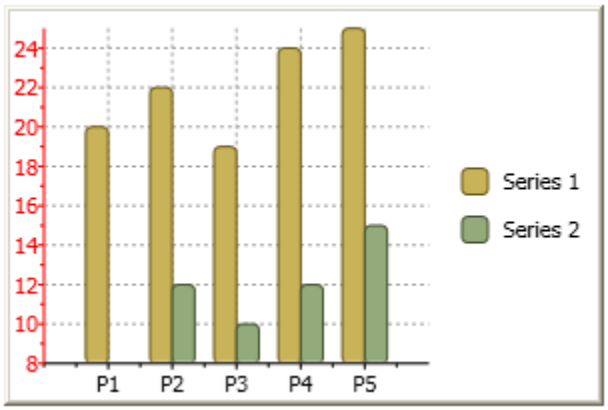
There are two types of ticks on the axis: major tick has a small line and corresponding label while the minor tick has only the line across the axis.

By default, the distance between ticks is calculated automatically.

To set a specific distance, use the MajorUnit and MinorUnit properties.

Default Ticks

The following image displays the default ticks:



Custom Ticks

The following chart image uses the MajorUnit and MinorUnit properties to set the specific distance, for example:

- Visual Basic

```
c1Chart1.View.AxisY.MajorUnit = 5
```

```
  c1Chart1.View.AxisY.MinorUnit = 1
```

- C#

```
c1Chart1.View.AxisY.MajorUnit = 5;
  c1Chart1.View.AxisY.MinorUnit = 1;
```

Time Axis

For time axis you can specify the MajorUnit and MinorUnit as a TimeSpan value:

- Visual Basic

```
c1Chart1.View.AxisY.MajorUnit = TimeSpan.FromHours(12)
```

- C#

```
c1Chart1.View.AxisY.MajorUnit = TimeSpan.FromHours(12);
```

Displaying Axis and Annotations on the Opposite Side of the Chart

To display the horizontal axis and annotations on the opposite side of the chart you can use the auxiliary axis and position the axis at the top with the title only like the following code:

- Visual Basic

```
c1Chart1.View.Axes.Add(new Axis()
      {
        AxisType = AxisType.X,
        Position = AxisPosition.Far,
        ItemsSource = new string[] { ""},
        Title = "Axis title",
      });
```

- C#

```
c1Chart1.View.Axes.Add(new Axis()
      {
        AxisType = AxisType.X,
        Position = AxisPosition.Far,
        ItemsSource = new string[] { ""},
        Title = "Axis title",
      });
```

Binding the Chart to a DataTable from DataSet

Here is sample code that creates the chart from the data table.

In Code:

- Visual Basic

```
Private _dataSet As DataSet

Private Sub Window_Loaded(ByVal sender As Object, ByVal e As
RoutedEventArgs)
    ' create connection and fill data set
    Dim mdbFile As String = "c:\db\nwind.mdb"
    Dim connString As String =
String.Format("Provider=Microsoft.Jet.OLEDB.4.0; Data Source={0}",
mdbFile)
    Dim conn As New OleDbConnection(connString)
```

```
    Dim adapter As New OleDbDataAdapter("SELECT TOP 10 ProductName,
UnitPrice FROM Products " & vbCr & vbLf & " ORDER BY UnitPrice;", conn)

    _dataSet = New DataSet()
    adapter.Fill(_dataSet, "Products")

    ' set data table rows as the source for chart data
    c1Chart1.Data.ItemsSource = _dataSet.Tables("Products").Rows
End Sub
```

- C#

```
DataSet _dataSet;

private void Window_Loaded(object sender, RoutedEventArgs e)
{
  // create connection and fill data set
  string mdbFile = @"c:\db\nwind.mdb";
  string connString = string.Format(
    "Provider=Microsoft.Jet.OLEDB.4.0; Data Source={0}",
    mdbFile);
  OleDbConnection conn = new OleDbConnection(connString);
  OleDbDataAdapter adapter = new OleDbDataAdapter(
     @"SELECT TOP 10 ProductName, UnitPrice FROM Products
       ORDER BY UnitPrice;", conn);

  _dataSet = new DataSet();
  adapter.Fill(_dataSet, "Products");

  // set data table rows as the source for chart data
  c1Chart1.Data.ItemsSource = _dataSet.Tables["Products"].Rows;
}
```

In XAML:

```
<c1chart:C1Chart.Data>
  <c1chart:ChartData ItemNameBinding="{Binding Path=[ProductName]}">
    <c1chart:DataSeries ValueBinding="{Binding Path=[UnitPrice]}"/>
  </c1chart:ChartData>
</c1chart:C1Chart.Data>
```

Customizing Chart

The following topics show how to modify the chart's data labels, plot element colors, and chart legend.

Changing Plot Element Colors

To change the colors assigned to the plot elements such as bars and pies(depending on chart type), you can either change the Palette property to one of the predefined color palettes or you can create a custom palette, such as:

```
Brush[] customBrushes = new Brush[2] { Brushes.Blue, Brushes.Orange };
c1Chart1.CustomPalette = customBrushes;
```

Hiding the Chart Legend

To programatically hide the chart legend you can do the following:

Add name to the legend in xaml and then you can change its visibility in code: legend.Visibility = ...

```
<c1chart:C1Chart x:Name="chart" >
    <c1chart:C1ChartLegend x:Name="legend"  />
    ...
 </c1chart:C1Chart>
```

Orienting the Data in the ChartLegend

To orient the chart legend horizontally, use the following code:

```
C1ChartLegend.Orientation = Horizontal
```

Showing Data Labels on the First of Each Month

To only show the data labels on the first of each month, use the following code:

- Visual Basic

```
c1Chart1.View.AxisX.IsTime = True
c1Chart1.View.AxisX.AnnoFormat = "MMM-dd"
' when MajorUnit=31 for time axis chart should
' take into account variable number of day in month
' and mark the first day of each month
c1Chart1.View.AxisX.MajorUnit = 31
```

- C#

```
c1Chart1.View.AxisX.IsTime = true;
c1Chart1.View.AxisX.AnnoFormat = "MMM-dd";
// when MajorUnit=31 for time axis chart should
// take into account variable number of day in month
// and mark the first day of each month
c1Chart1.View.AxisX.MajorUnit = 31;
```

Adding a Chart Label

To add a label above your chart, add a **TextBlock** element after the opening c1chart:C1Chart tag and enter the title for your chart in the **Text** property:

```
<TextBlock DockPanel.Dock="Top" Text="Chart Title"
HorizontalAlignment="Center"/>
```

Bar/Column Chart Tasks

The following topics show how to customize the appearance and behavior of the Bar/Column charts.

Changing the Corners of the Rectangles in Bar/Column Charts

Bars/columns do not have rounded corners by default. The radius of rectangle corners can be set using Bar class, for example:

```
ds.Symbol = new Bar() { RadiusX=5, RadiusY=5};
```

Creating a Mouse Click Event for a Column Chart

You can add animation when you click on any column in the Column chart, using **MouseDown** and **MouseLeave** events, like the following XAML code:

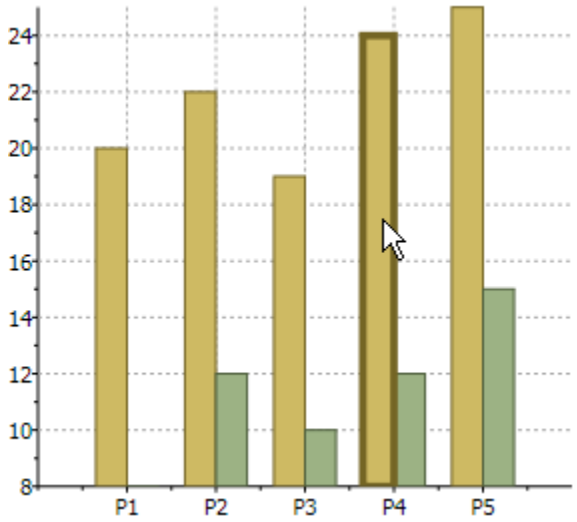
```
<Window x:Class="MouseEvent.Window1"
```

```
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
          xmlns:sys="clr-namespace:System;assembly=mscorlib"
    Title="Window1" Height="300" Width="300"
xmlns:c1chart="http://schemas.componentone.com/xaml/c1chart"
Loaded="Window_Loaded">
    <Grid>
        <Grid.Resources>
            <Style x:Key="sstyle" TargetType="{x:Type
c1chart:PlotElement}">
                <Setter Property="StrokeThickness" Value="1" />
                <Setter Property="Canvas.ZIndex" Value="0" />
                <Style.Triggers>
                    <EventTrigger
RoutedEvent="c1chart:PlotElement.MouseDown">
                        <BeginStoryboard>
                            <Storyboard>
                                <Int32Animation
Storyboard.TargetProperty="(Panel.ZIndex)"
                                 To="1" />
                                <DoubleAnimation
Storyboard.TargetProperty="StrokeThickness"
                                To="4" Duration="0:0:0.3"
                                  AutoReverse="True"
                                 RepeatBehavior="Forever" />
                                </Storyboard>
                        </BeginStoryboard>
                    </EventTrigger>
                    <EventTrigger
RoutedEvent="c1chart:PlotElement.MouseLeave">
                        <BeginStoryboard>
                            <Storyboard>
                                <DoubleAnimation
Storyboard.TargetProperty="StrokeThickness" />
                                <Int32Animation
Storyboard.TargetProperty="(Panel.ZIndex)" />
                                </Storyboard>
                        </BeginStoryboard>
                    </EventTrigger>
                </Style.Triggers>
            </Style>
        </Grid.Resources>
        <c1chart:C1Chart Margin="0" Name="c1Chart1" ChartType="Column">
            <c1chart:C1Chart.Data>
                <c1chart:ChartData>
                    <c1chart:ChartData.ItemNames>P1 P2 P3 P4
P5</c1chart:ChartData.ItemNames>
                    <c1chart:DataSeries SymbolStyle="{StaticResource
sstyle}" Values="20
22 19 24 25" />
                    <c1chart:DataSeries SymbolStyle="{StaticResource
sstyle}" Values="8
12 10 12 15" />
                </c1chart:ChartData>
            </c1chart:C1Chart.Data>
        </c1chart:C1Chart>
    </Grid>
```

```
</Window>
```

This Topic Illustrates the Following:

Click on any of the columns and notice the animation around the borders of the rectangles:



Specifying the Color of Each Bar/Column in the Data Series

You can specify the color of each bar/column in the datat series PlotElementLoaded event using the following code:

```
var palette = new Brush[] { Brushes.Red, Brushes.Plum, Brushes.Purple
};
      dataSeries.PlotElementLoaded += (s, e) =>
        {
          PlotElement pe = (PlotElement)s;
          if (pe.DataPoint.PointIndex >= 0)
            pe.Fill = palette[pe.DataPoint.PointIndex %
palette.Length];
        };
```

Candle Tasks

The following tasks show how to customize Candle charts.

Changing the Candle Stick Width

To change the candle stick width, use the SymbolSize property like the following:

```
ds.SymbolSize = new Size(5, 5);
```

Runtime Tasks

The following tasks show how to modify the chart's behavior during runtime.

Changing Rotation for 3D Chart

To change the rotation view for 3D chart type at runtime, add the Rotate3DAction class to Actions collection. For example, to rotate chart with the middle mouse button use the following XAML code:

```
<c1chart:C1Chart.Actions>
   <c1chart:Rotate3DAction MouseButton="Middle" />
</c1chart:C1Chart.Actions>
```

Enabling Run-Time Interaction for the 2D Cartesian Chart

The action for zooming, scaling, and translating is invoked by the specified mouse button with optional keyboard modifiers (Alt|Ctrl|Shift). The actions should be placed in the Actions collection. The following XAML code below defines a set of actions.

```
<c1chart:C1Chart.Actions>
<!-- use left mouse button to scroll through data -->
<c1chart:TranslateAction MouseButton="Left" />
<!-- use ctrl+left mouse button to change scale -->
<c1chart:ScaleAction MouseButton="Left" Modifiers="Ctrl"/>
<!-- use shift+left mouse to zoom selected rectangular area-->
<c1chart:ZoomAction MouseButton="Left" Modifiers="Shift" />
</c1chart:C1Chart.Actions>
```

The actions are closely related with Axis properties(**Min**, **Max**, **Scale**, **MinScale**). When **Axis.Scale**=1 the translate action is not available along the axis. The MinScale sets limitation of zoom or scale that can be achieved during action.

Scaling a Bubble Chart While Zooming

To scale the Bubble chart while zooming, adjust the scale in the PlotElementLoaded event like the following:

```
var ds = new BubbleSeries()
{
XValuesSource = new double[] { 1, 2, 3, 4 },
ValuesSource = new double[] { 1, 2, 3, 4 },
SizeValuesSource = new double[] { 1, 2, 3, 4 },
};

ds.PlotElementLoaded += (s, e) =>
{
var pe = (PlotElement)s;
pe.RenderTransform = new ScaleTransform()
{
ScaleX = 1.0 / chart.View.AxisX.Scale,
ScaleY = 1.0 / chart.View.AxisY.Scale
};
pe.RenderTransformOrigin = new Point(0.5, 0.5);
};

chart.Data.Children.Add(ds);
chart.ChartType = ChartType.Bubble;

chart.Actions.Add(new TranslateAction());
chart.Actions.Add(new ScaleAction() { Modifiers = ModifierKeys.Control
});
```

Scaling Both Independent Axes

To scale both independent axes you should link the scale and value properties of both axes using the PropertyChanged event like the following:

```
 // suppose ay2 is auxiliary y-axis

 ((INotifyPropertyChanged)chart.View.AxisY).PropertyChanged += (s, e)
=>
   {
     if (e.PropertyName == "Scale")
     {
       ay2.Scale = chart.View.AxisY.Scale;
     }
     else if (e.PropertyName == "Value")
     {
       ay2.Value = chart.View.AxisY.Value;
     }
   };
```

Swapping X and Y Axes During Runtime

To invert the axes after the chart was loaded, use the following code:

```
((Renderer2D)c1Chart1.Data.Renderer).Inverted = true;
```

Zooming in C1Chart

To add zooming behavior in C1Chart, use some custom code in the Chart's MouseWheel event.

```
private void chart_MouseWheel(object sender, MouseWheelEventArgs e)
{
      if (Keyboard.Modifiers == ModifierKeys.Control && e.Delta == -
120)
      {
       chart.View.AxisX.Scale += .1;
       chart.View.AxisY.Scale += .1;
       }
else if (Keyboard.Modifiers == ModifierKeys.Control && e.Delta == 120)
      {
          chart.View.AxisX.Scale -= .1;
          chart.View.AxisY.Scale -= .1;
          }
}
```

To enable the user to move the chart around when it's zoomed, add the following to C1Chart's XAML:

```
 <c1c:C1Chart x:Name="chart" MouseWheel="chart_MouseWheel" >
    <c1c:C1Chart.Actions>
        <c1c:TranslateAction MouseButton="Left" />
    </c1c:C1Chart.Actions>
</c1c:C1Chart>
```

Creating Different Chart Types

The following tasks show how to add different chart types.

Adding a Bar Series and a Line Series at the Same Time

To add a Bar series and a Line series programmatically, use the following code:

```
chart.Data.Children.Add(new XYDataSeries() {
       ChartType=ChartType.Column,
       XValuesSource = new double[] {1,2,3 },
       ValuesSource = new double[] {1,2,3 } });

     chart.Data.Children.Add(new XYDataSeries() {
       ChartType = ChartType.Line,
       XValuesSource = new double[] { 1, 2, 3 },
       ValuesSource = new double[] { 3, 2, 1 } });
```

Creating Combinations of Charts

Using different templates for the different data series it is easy to create various combinations of chart types.

Column-line chart

This chart can be created with DataSeries.ChartType.

```
<c1chart:C1Chart Name="c1chart1">
   <c1chart:C1Chart.Data>
     <c1chart:ChartData >
       <!-- Default(column) appearance for the first series -->
       <c1chart:DataSeries Label="series 1" Values="0.5 2 3 4" />
       <!-- Second series stars connected with lines-->
       <c1chart:DataSeries Label="series 2" Values="1 3 2 1"
        ChartType="LineSymbols" SymbolMarker="Star4" />
     </c1chart:ChartData>
   </c1chart:C1Chart.Data>
</c1chart:C1Chart>
```

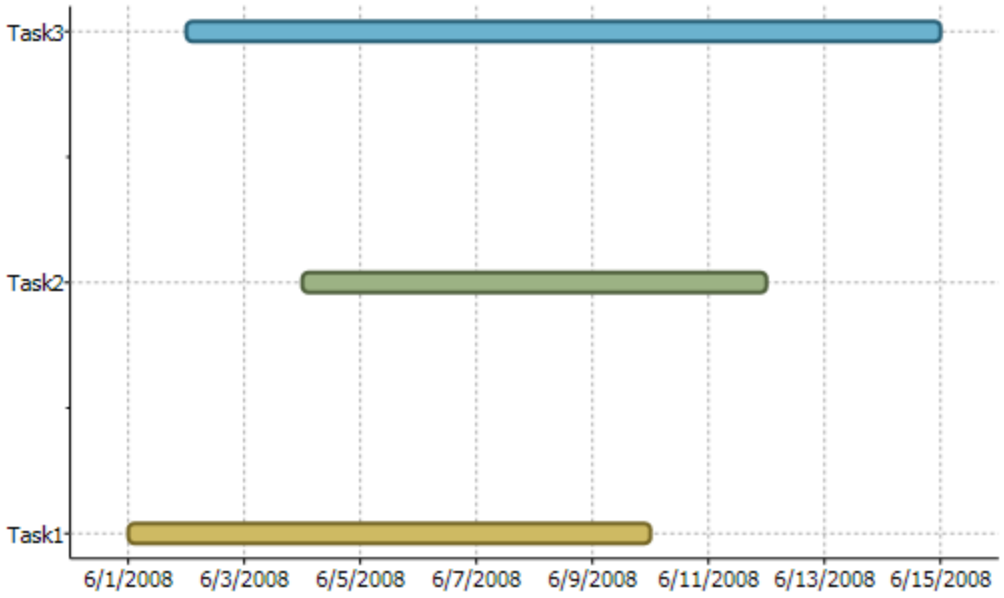
Creating a Gantt Chart

To create a Gantt chart, use the following XAML code:

```
<c1chart:C1Chart Margin="0" Name="c1Chart1"
            xmlns:sys="clr-namespace:System;assembly=mscorlib">
  <c1chart:C1Chart.Resources>
    <x:Array x:Key="start" Type="sys:DateTime" >
      <sys:DateTime>2008-6-1</sys:DateTime>
      <sys:DateTime>2008-6-4</sys:DateTime>
      <sys:DateTime>2008-6-2</sys:DateTime>
    </x:Array>
    <x:Array x:Key="end" Type="sys:DateTime">
      <sys:DateTime>2008-6-10</sys:DateTime>
      <sys:DateTime>2008-6-12</sys:DateTime>
      <sys:DateTime>2008-6-15</sys:DateTime>
    </x:Array>
  </c1chart:C1Chart.Resources>
```

```
  <c1chart:C1Chart.Data>
    <c1chart:ChartData>
      <c1chart:ChartData.Renderer>
        <c1chart:Renderer2D Inverted="True" ColorScheme="Point"/>
      </c1chart:ChartData.Renderer>
      <c1chart:ChartData.ItemNames>Task1 Task2
Task3</c1chart:ChartData.ItemNames>
      <c1chart:HighLowSeries HighValuesSource="{StaticResource end}"
                             LowValuesSource="{StaticResource start}"/>
    </c1chart:ChartData>
  </c1chart:C1Chart.Data>
  <c1chart:C1Chart.View>
    <c1chart:ChartView>
      <c1chart:ChartView.AxisX>
        <c1chart:Axis IsTime="True" AnnoFormat="d"/>
      </c1chart:ChartView.AxisX>
    </c1chart:ChartView>
  </c1chart:C1Chart.View>
</c1chart:C1Chart>
```

Run your project and the data in the XAML code produces the following **Gantt** chart in your Window:



Creating a Gaussian Curve

To create a **Gaussian Curve** in C1Chart, use the following code:

```
// create and add to the chart data series representing Gaussian function
  //   y(x) = a * exp( -(x-b)*(x-b) / (2*c*c))
  // in the interval from x1 to x2
      void CreateGaussian(double x1, double x2, double a, double b,
double c)
      {
```

```
// number of points
      int cnt = 200;       var xvals = new double[cnt];
      var yvals = new double[cnt];

      double dx = (x2 - x1) / (cnt-1);

      for (int i = 0; i < cnt; i++)
      {
        var x = x1 + dx * i;
        xvals[i] = x;
        x =  (x - b) / c;
        yvals[i] = a * Math.Exp(-0.5*x*x);
      }

      var ds = new XYDataSeries()
      {
        XValuesSource = xvals,
        ValuesSource = yvals,
        ChartType = ChartType.Line
      };

      chart.Data.Children.Add(ds);
    }
```

Creating a HiLoOpenClose Chart

To programmatically create a **HiLoOpenClose** chart, use the following code:

```
HighLowOpenCloseSeries ds = new HighLowOpenCloseSeries()
{
XValueBinding = new System.Windows.Data.Binding("NumberOfDay"),
HighValueBinding = new System.Windows.Data.Binding("High"),
LowValueBinding = new System.Windows.Data.Binding("Low"),
OpenValueBinding = new System.Windows.Data.Binding("Open"),
CloseValueBinding = new System.Windows.Data.Binding("Close"),
SymbolStrokeThickness = 1,     SymbolSize = new Size(5, 5)
}
ds.PlotElementLoaded += (s, e) =>
{
PlotElement pe = (PlotElement)s;
double open = (double)pe.DataPoint["OpenValues"];
double close = (double)pe.DataPoint["CloseValues"];
if (open > close)
{
pe.Fill = green;
pe.Stroke = green;
}
else
{
pe.Fill = red;
pe.Stroke = red;
}
};
```

Creating a Pareto Chart or Scatter Chart

To create a **Pareto** or **Scatter** chart, use the following XAML code:

```
<c1chart:C1Chart Name="c1Chart1">
   <c1chart:C1Chart.View>
     <c1chart:ChartView>
       <c1chart:ChartView.AxisX>
         <c1chart:Axis AnnoAngle="-75" MajorGridStroke="Gray"/>
       </c1chart:ChartView.AxisX>
       <!-- Standard(default) left y-axis -->
       <c1chart:ChartView.AxisY>
         <c1chart:Axis Min="0" Max="50" Title="Frequency"
MajorGridStroke="Gray"/>
       </c1chart:ChartView.AxisY>
       <!-- Auxiliary(right) y-axis -->
       <c1chart:Axis Name="ay2" AxisType="Y" Position="Far"
AnnoFormat="p"
           Min="0" Max="1" />
     </c1chart:ChartView>
   </c1chart:C1Chart.View>
   <c1chart:C1Chart.Data>
     <c1chart:ChartData>
       <c1chart:ChartData.ItemNames>Documents Quality Packaging
Delivery Other</c1chart:ChartData.ItemNames>
       <c1chart:DataSeries Values="40 30 20 5 5" />
       <c1chart:DataSeries AxisY="ay2" Values="0.4 0.7 0.9 0.95 1.0"
ChartType="LineSymbols" />
     </c1chart:ChartData>
   </c1chart:C1Chart.Data>
</c1chart:C1Chart>
```

Creating a Stacked Area Chart

To create stacked chart you should set ChartType instead of ChartType like the following code example:

```
<c1:C1Chart ChartType="AreaStacked" >
 <c1:C1Chart.Data>
  <c1:ChartData ItemNames="P1 P2 P3 P4 P5">
   <c1:DataSeries Label="Series 1" Values="20 22 19 24 25" />
   <c1:DataSeries Label="Series 2" Values="8 12 10 12 15" />
  </c1:ChartData>
 </c1:C1Chart.Data>
 <c1:C1ChartLegend DockPanel.Dock="Right" />
</c1:C1Chart>
```

Pie Tasks

The following topics show how to connect lines to prevent pie overlapping, add labels to pie charts, change the offset for all pie slices, and set the default viewing angle for the 3D Pie chart.

Adding Connecting Lines to Prevent Pie Overlapping

You can add connecting lines with the **PlotElement.LabelLine Attached** property like the following XAML code:

```
<c1:DataSeries.PointLabelTemplate>
  <DataTemplate>
    <Border BorderBrush="DarkGray" BorderThickness="1"
Background="LightGray"
            c1:PlotElement.LabelAlignment="Auto"
            c1:PlotElement.LabelOffset="30,0">
      <TextBlock Text="{Binding Value, StringFormat=0}" />
      <c1:PlotElement.LabelLine>
        <Line Stroke="LightGray" StrokeThickness="2" />
      </c1:PlotElement.LabelLine>
    </Border>
  </DataTemplate>
</c1:DataSeries.PointLabelTemplate>
```

Adding Labels to Pie Charts

To add multiple values to a **Pie** chart label you can create a label template like the following:

```
  <c1chart:C1Chart Name="c1Chart1" ChartType="Pie">
      <c1chart:C1Chart.Resources>
        <DataTemplate x:Key="lbl">
          <StackPanel>
            <StackPanel Orientation="Horizontal">
              <TextBlock Text="{Binding Path=Name}" />
              <TextBlock Text="=" />
              <TextBlock Text="{Binding Path=Value}" />
            </StackPanel>
            <TextBlock Text="{Binding
Path=PercentageSeries,Converter={x:Static c1chart:Converters.Format},
ConverterParameter=#.#%}"/>
          </StackPanel>
        </DataTemplate>
      </c1chart:C1Chart.Resources>
      <c1chart:C1Chart.Data>
        <c1chart:ChartData>
          <c1chart:ChartData.ItemNames>P1 P2 P3 P4
P5</c1chart:ChartData.ItemNames>
          <c1chart:DataSeries Values="20 22 19 24 25"
PointLabelTemplate="{StaticResource lbl}" />
        </c1chart:ChartData>
      </c1chart:C1Chart.Data>
      <c1chart:C1ChartLegend DockPanel.Dock="Right" />
    </c1chart:C1Chart>
```

Changing the Offset for All Slices

To change the offset for all Pie slices, use the following code:

```
chart.DataContext = new double[] { 1, 2, 3 };
      chart.ChartType = ChartType.Pie;
      chart.Loaded += (s, e) =>
((BasePieRenderer)chart.Data.Renderer).Offset = 0.1;
```

Also it's possible to change offset for the specific slice but it requires to change the position manually in PlotElementLoaded event.

Setting the Default Viewing Angle for 3D Pie Chart

To set the default viewing angle for the 3D pie chart, use the following code:

```
chart.View.Camera.Transform = new RotateTransform3D(new
AxisAngleRotation3D(new Vector3D(0,0,1),45));
```

Disabling Chart Optimization After it has been Set

To disable the chart optimization once it has been set like the following:

- Visual Basic
```
LineAreaOptions.SetOptimizationRadius(c1Chart1, 2.0)
```
- C#
```
LineAreaOptions.SetOptimizationRadius(c1Chart1, 2.0);
```

You can set it to the default value, NaN, like the following:

- Visual Basic
```
LineAreaOptions.SetOptimizationRadius(c1Chart1, double.NaN)
```
- C#
```
LineAreaOptions.SetOptimizationRadius(c1Chart1, double.NaN);
```

Displaying Gaps in Line or Area Charts

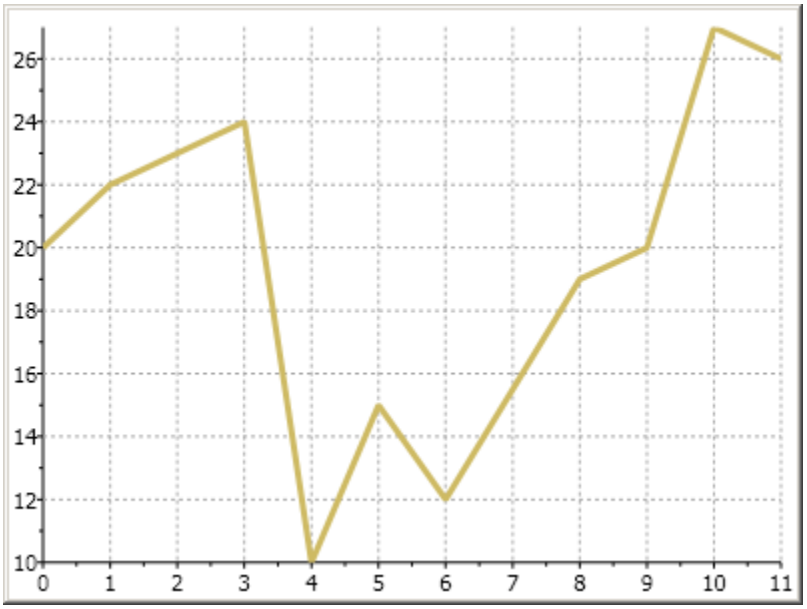
By default, if there is a hole(double.NaN) in data values chart just skips the value and draws line to next valid data point.

To change this behavior and show the gaps at the hole values, set Display = ShowNaNGap.

For example, the following XAML code includes specified holes in the DataSeries:

```
<c1chart:C1Chart  Name="c1Chart1" ChartType="Line">
     <c1chart:C1Chart.Data>
       <c1chart:ChartData>
         <c1chart:DataSeries Values="20 22 NaN 24 15 NaN 27 26"
           ConnectionStrokeThickness="3" />
       </c1chart:ChartData>
     </c1chart:C1Chart.Data>
   </c1chart:C1Chart>
```

The chart appears similar to the following without the Display property set:



To show a gap between the chart lines in a **Line** chart you can set the **Display** property to ShowNaNGap like the following:

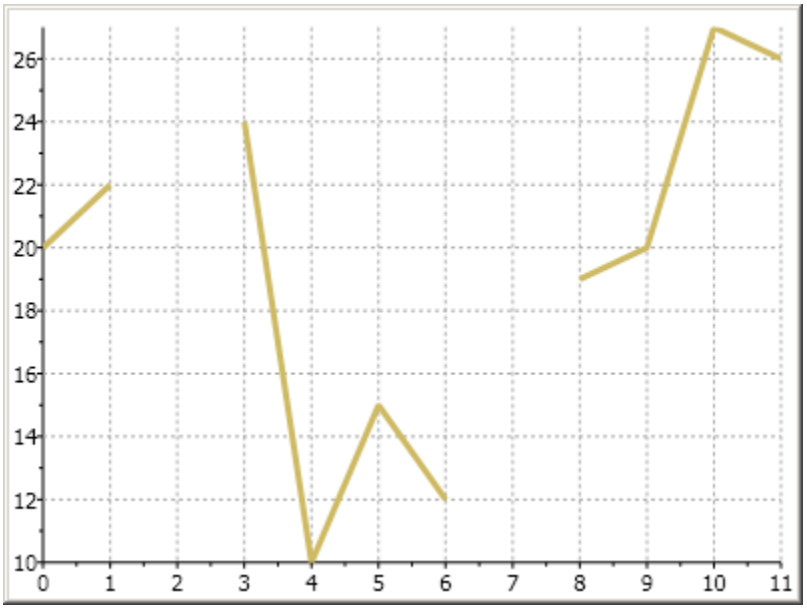
- Visual Basic

```
Me.C1Chart1.Data.Children(1).Display =
C1.WPF.C1Chart.SeriesDisplay.ShowNaNGap
```

- C#

```
this.C1Chart1.Data.Children[1].Display =
C1.WPF.C1Chart.SeriesDisplay.ShowNaNGap;
```

The line chart will display a gap between the chart lines, similar to the following:



Performing Batch Updates

You can perform batch updates without refreshing the chart after each change by entering your code inside the **BeginUpdate()/EndUpdate()** methods like the following:

- Visual Basic

```
C1Chart1.BeginUpdate()
    ' change or format the chart, add data etc.
    ...
    C1Chart1.EndUpdate()
```

- C#

```
c1Chart1.BeginUpdate();
    // change or format the chart, add data etc.
    ...
    c1Chart1.EndUpdate();
```

Saving and Exporting C1Chart

The following tasks show how to save and export chart to different formats.

Exporting Chart into a PDF Format

To export chart to the bitmap image and create the PDF with the image using C1Pdf library, use the following code:

```
// save chart image to stream
            MemoryStream ms = new MemoryStream();
            chart.SaveImage(ms, ImageFormat.Png);

    // create image instance from stream
            var img = System.Drawing.Image.FromStream(ms);
```

```
// create and save pdf document
        C1PdfDocument pdf = new C1PdfDocument();
        pdf.DrawImage( img, new
      System.Drawing.RectangleF(0,0,img.Width,img.Height));
        pdf.Save("doc.pdf");
```

Exporting Chart Image

You can export a chart image by using the **RenderTargetBitmap** like in the following code:

- Visual Basic

```
Dim bm As New RenderTargetBitmap(CInt(c1Chart1.ActualWidth),
CInt(c1Chart1.ActualHeight), 96, 96, PixelFormats.[Default])
bm.Render(c1Chart1)

Dim enc As New PngBitmapEncoder()
enc.Frames.Add(BitmapFrame.Create(bm))

Dim fs As New FileStream("chart.png", FileMode.Create)
enc.Save(fs)
```

- C#

```
RenderTargetBitmap bm = new RenderTargetBitmap(
      (int)c1Chart1.ActualWidth,(int)c1Chart1.ActualHeight,
        96, 96, PixelFormats.Default);
bm.Render(c1Chart1);

PngBitmapEncoder enc = new PngBitmapEncoder();
enc.Frames.Add(BitmapFrame.Create(bm));

FileStream fs = new FileStream("chart.png", FileMode.Create);
enc.Save(fs);
```

Saving C1Chart as a .Png File

To save **C1Chart** as a .Png file, use the following code:

- Visual Basic

```
' save image to file
Using stm = System.IO.File.Create("chart.png")
    c1Chart1.SaveImage(stm, C1.WPF.C1Chart.Extended.ImageFormat.Png)
End Using
```

- C#

```
// save image to file
      using (var stm = System.IO.File.Create("chart.png"))
      {
        c1Chart1.SaveImage(stm, C1.WPF.C1Chart.Extended.ImageFormat.Png);
      }
```