ComponentOne

# Chart for Silverlight

*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)



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

# Table of Contents

# ComponentOne Chart for Silverlight Overview

Transform your data and add professional-grade charts to your Silverlight apps with one setting; **ComponentOne Chart™ for Silverlight** supports all popular chart types including bar, column, line, area, pie, and more. Take your data visualization to the next level today.

For a list of the latest features added to **ComponentOne Studio for Silverlight**, visit What's New in Studio for Silverlight.

**Getting Started**

Get started with the following topics:

- Key Features (page 17)
- Quick Start (page 19)
- Task-Based Help (page 141)

## Installing Chart for Silverlight

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

### Chart for Silverlight Setup Files

The **ComponentOne Chart for Silverlight** installation program will create the following directory: **C:\Program Files\ComponentOne\Studio for Silverlight 4.0**. This directory contains the following subdirectories:

| | |
|---|---|
| **Bin** | Contains copies of ComponentOne binaries (DLLs, EXEs, design-time assemblies). |
| **Help** | Contains documentation for all Studio components and other useful resources including XAML files. |

**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 Vista/Windows 7 machines:

**Windows XP path:** C:\Documents and Settings\<username>\My Documents\ComponentOne Samples\Studio for Silverlight 4.0

**Windows Vista and Windows 7 path:** C:\Users\<username>\Documents\ComponentOne Samples\Studio for Silverlight 4.0

See the Chart for Silverlight Samples (page 139) topic for more information about each sample.

## System Requirements

System requirements for **ComponentOne Chart for Silverlight** include the following:

1. Microsoft Silverlight 4.0 or later
2. Microsoft Visual Studio 2008 or later

## Installing Demonstration Versions

If you wish to try **ComponentOne Chart for Silverlight** 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 the registered version will stamp every application you compile so a ComponentOne banner will not appear when your users run the applications.

## Uninstalling Chart for Silverlight

To uninstall **ComponentOne Chart for Silverlight**:

1. Open the Control Panel and select Add or Remove Programs (XP) or Programs and Features (Windows 7/Vista).
2. Select ComponentOne Studio for Silverlight 4.0 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, frequently asked licensing questions, and the ComponentOne licensing model, is available online at http://www.componentone.com/SuperPages/Licensing/.

# Licensing FAQs

The **ComponentOne Studio for Silverlight** product is a commercial product. It is not shareware, freeware, or open source. If you use it in production applications, please purchase a copy from our Web site or from the software reseller of your choice.

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 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.

### Studio for Silverlight Licensing

Licensing for **ComponentOne Studio for Silverlight** is similar to licensing in other ComponentOne products but there are a few differences to note.

Initially licensing in handled similarly to other ComponentOne products. 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.

In **ComponentOne Studio for Silverlight**, when a control is dropped on a form, a license nag dialog box appears one time.

The **About** dialog box displays version information, online resources, and (if the control is unlicensed) buttons to purchase, activate, and register the product.

All ComponentOne products are designed to display licensing information at run time if the product is not licensed. None will throw licensing exceptions and prevent applications from running. Each time an unlicensed Silverlight application is run; end-users will see the pop-up dialog box.

To stop this message from appearing, enter the product's serial number by clicking the **Activate** button on the **About** dialog box of any ComponentOne product, if available, or by rerunning the installation and entering the serial number in the licensing dialog box. To open the **About** dialog box, right-click the control and select the **About** option:

Note that when the user modifies any property of a ComponentOne Silverlight control in Visual Studio or Blend, the product will check if a valid license is present. If the product is not currently licensed, an attached property will be added to the control (the **C1NagScreen.Nag** property). Then, when the application executed, the product will check if that property is set, and show a nag screen if the **C1NagScreen.Nag** property is set to **True**. If the user has a valid license the property is not added or is just removed.

One important aspect of this of this process is that the user should manually remove all instances of **c1:C1NagScreen.Nag="true"** in the XAML markup in all files after registering the license (or re-open all the files that include ComponentOne controls in any of the editors). This will ensure that the nag screen does not appear when the application is run.

# 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 Silverlight** 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.Silverlight.dll
- C1.Silverlight.Chart.dll
- C1.Silverlight.Chart.Editor.dll
- C1.Silverlight.Chart.Extended.dll

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

# About This Documentation

You can create your 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. In this documentation, we will use the **Design** workspace of Blend for most examples.

**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.

# Silverlight Resources

This help file focuses on **ComponentOne Studio for Silverlight**. For general help on getting started with Silverlight, we recommend the following resources:

- http://www.silverlight.net

  The official Silverlight site, with many links to downloads, samples, tutorials, and more.

- http://silverlight.net/learn/tutorials.aspx

  Silverlight tutorials by Jesse Liberty. Topics covered include:

  - Tutorial 1: Silverlight User Interface Controls
  - Tutorial 2: Data Binding
  - Tutorial 3: Displaying SQL Database Data in a DataGrid using LINQ and WCF
  - Tutorial 4: User Controls
  - Tutorial 5: Styles, Templates and Visual State Manager
  - Tutorial 6: Expression Blend for Developers
  - Tutorial 7: DataBinding & DataTemplates Using Expression Blend
  - Tutorial 8: Multi-page Applications
  - Tutorial 9: ADO.NET DataEntities and WCF Feeding a Silverlight DataGrid
  - Tutorial 10: Hyper-Video

- http://timheuer.com/blog/articles/getting-started-with-silverlight-development.aspx

  Silverlight tutorials by Tim Heuer. Topics covered include:

  - Part 1: Really getting started – the tools you need and getting your first Hello World
  - Part 2: Defining UI Layout – understanding layout and using Blend to help
  - Part 3: Accessing data – how to get data from where
  - Part 4: Binding the data – once you get the data, how can you use it?
  - Part 5: Integrating additional controls – using controls that aren't a part of the core
  - Part 6: Polishing the UI with styles and templates
  - Part 7: Taking the application out-of-browser

- http://weblogs.asp.net/scottgu/pages/silverlight-posts.aspx

Scott Guthrie's Silverlight Tips, Tricks, Tutorials and Links Page. A useful resource, this page links to several tutorials and samples.

- http://weblogs.asp.net/scottgu/archive/2008/02/22/first-look-at-silverlight-2.aspx

  An excellent eight-part tutorial by Scott Guthrie, covering the following topics:

  - Part 1: Creating "Hello World" with Silverlight 2 and VS 2008
  - Part 2: Using Layout Management
  - Part 3: Using Networking to Retrieve Data and Populate a DataGrid
  - Part 4: Using Style Elements to Better Encapsulate Look and Feel
  - Part 5: Using the ListBox and DataBinding to Display List Data
  - Part 6: Using User Controls to Implement Master/Details Scenarios
  - Part 7: Using Templates to Customize Control Look and Feel
  - Part 8: Creating a Digg Desktop Version of our Application using WPF
- http://blogs.msdn.com/corrinab/archive/2008/03/11/silverlight-2-control-skins.aspx

  A practical discussion of skinning Silverlight controls and applications by Corrina Barber.

# Creating a .NET Project in Visual Studio

Complete the following steps to create a new Silverlight project in Microsoft Visual Studio 2010:

1. Select **File | New | Project** to open the **New Project** dialog box in Visual Studio 2010.
2. In the **Project types** pane, expand either the **Visual Basic** or **Visual C#** node and select **Silverlight**.
3. Choose **Silverlight Application** in the **Templates** pane.
4. Name the project, specify a location for the project, and click **OK**.

   Next, Visual Studio will prompt you for the type of hosting you want to use for the new project.
5. In the **NewSilverlight Application** dialog box, select **OK** to accept the default name and options and to create the project.

# Creating a Silverlight Project in Expression Blend

Complete the following steps to create a new Silverlight project in Microsoft Expression Blend 4:

1. Select **File | New Project** to open the **New Project** dialog box in Blend 4.
2. In the **Project types** pane, click the **Silverlight** node.
3. In the right pane, choose **Silverlight Application + Website** in the **Templates** pane to create a project with an associated Web site.
4. Name the project, specify a location for the project, choose a language (**Visual C#** or **Visual Basic**), and click **OK**.

   Your new project will be created.

---

# Adding the Chart for Silverlight Components to a Blend Project

**To add a reference to the assembly:**

1. Select **Project | Add Reference**.

1. Browse to find the **C1.Silverlight.Chart.dll** assembly installed with **Chart for Silverlight**.

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

2. Select **C1.Sivlerlight.Chart.dll** and click **Open**. A reference is added to your project.

**To add a component from the Asset Library:**

1. Once you have added references to the **C1.Silverlight.Chart.dll**, **C1.Silverlight.Chart.Extended.dll**, and **C1.Chart.Editor.dll** assemblies, click the **Asset Library** button in the Blend Toolbox. The **Asset Library** appears.
2. Click the **Controls** drop-down arrow and select **All**.
3. Select **C1Chart**. The component will appear in the Toolbox below the **Asset Library** button.
4. Double-click the **C1Chart** component in the Toolbox to add it to **Window1.xaml**.

# Adding the Chart for Silverlight Components to a Visual Studio Project

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

**ComponentOne Chart for Silverlight** provides the following controls:

- C1Chart
- C1ChartLegend

To use a **Chart for Silverlight** control, add it to the window or add a reference to the **C1.Silverlight.Chart** assembly to your project.

**Manually Adding Chart for Silverlight to the Toolbox**

When you install **Chart for Silverlight**, the following **Chart for Silverlight** control will appear in the Visual Studio Toolbox customization dialog box:

- C1Chart
- C1ChartLegend

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 Silverlight** components appear on its own tab in the Toolbox, select **Add Tab** from the context menu and type in the tab name, **C1Chart**, 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 **Silverlight Components** tab.
5. Sort the list by Namespace (click the *Namespace* column header) and select the check boxes for components belonging to the **C1.Silverlight** namespace. Note that there may be more than one component for each namespace.

**Adding Chart to the Window**

To add **ComponentOne Chart for Silverlight** 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 Silverlight** assembly, complete the following steps:

1. Select the **Add Reference** option from the **Project** menu of your project.
2. Select the **ComponentOne Chart for Silverlight** assembly from the list on the **.NET** tab or on the **Browse** tab, browse to find the **C1.Silverlight.Chart.dll** assembly 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.Silverlight.Chart
```

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

# Using Templates

The previous sections focused on the **ComponentOne Studio for Silverlight** controls. The following topics focus on Data and Control Templates, and how they are applied to Silverlight controls in general (including controls provided by Microsoft). If you are an experienced Silverlight developer, this information may be of no interest to you.

## Data Templates

**DataTemplates** are a powerful feature in Silverlight. They are virtually identical to the **DataTemplates** in WPF, so if you know WPF there's nothing really new about them.

On the other hand, if you have never used WPF and have seen pieces of XAML that contain styles and templates, you may be confused by the concepts and notation. The good news is DataTemplates are very powerful and are not overly complicated. Once you start using them, the concept will make sense in a couple of minutes and you will be on your way. Remember, just reading the tutorial probably won't be enough to fully grasp the concept. After reading, you should play with the projects.

### *Create the "Templates" Solution*

To illustrate the power of DataTemplates, let's create a new Silverlight solution. Call it "Templates". Complete the following steps:

1. Select **File | New | Project** to open the **New Project** dialog box in Visual Studio 2008.
2. In the **Project types** pane, expand either the **Visual Basic** or **Visual C#** node and select **Silverlight**.
3. Choose **Silverlight Application** in the **Templates** pane.
4. Name the project "Templates", specify a location for the project, and click **OK**.

   Next, Visual Studio will prompt you for the type of hosting you want to use for the new project.
5. In the **New Silverlight Application** dialog box, select **OK** to accept the default name ("Templates.Web") and settings and create the project.
6. Right-click the **Templates** project in the Solution Explorer and select **Add Reference**.
7. In the **Add Reference** dialog box locate and select the C1.Silverlight.dll assembly and click **OK** to add a reference to your project.

   This is required since we will be adding C1.Silverlight controls to the page.
8. Now, open the **MainPage.xaml** file in the **Templates** project and paste in the XAML below:

```
<UserControl x:Class="Templates.MainPage"
    xmlns="http://schemas.microsoft.com/client/2007"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

```
    xmlns:C1_Silverlight="clr-
namespace:C1.Silverlight;assembly=C1.Silverlight">

  <Grid x:Name="LayoutRoot" >
    <Grid.Background>
      <LinearGradientBrush EndPoint="0.5,1" StartPoint="0.5,0">
        <GradientStop Color="#FF7EB9F0"/>
        <GradientStop Color="#FF284259" Offset="1"/>
      </LinearGradientBrush>
    </Grid.Background>

    <!-- Grid layout -->
    <Grid.RowDefinitions>
      <RowDefinition Height="30" />
      <RowDefinition Height="*" />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition Width="*"/>
      <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>

    <!-- Page title -->
    <TextBlock Text="Silverlight Templates" Grid.Column="0"
Grid.ColumnSpan="2"
        TextAlignment="Center" FontSize="18" FontWeight="Bold" />

    <!-- ListBox on the left -->
    <StackPanel Grid.Row="1" Margin="5" >
      <TextBlock Text="ListBox Control" />
      <ListBox x:Name="_listBox" />
    </StackPanel>

    <!-- C1ComboBoxes on the right -->
    <StackPanel Grid.Column="2" Grid.Row="1" Margin="5" >
      <TextBlock Text="C1ComboBox Controls" />
      <C1_Silverlight:C1ComboBox x:Name="_cmb1" Margin="0,0,0,5" />
      <C1_Silverlight:C1ComboBox x:Name="_cmb2" Margin="0,0,0,5" />
    </StackPanel>

  </Grid>
</UserControl>
```

This creates a page with two columns. The left column has a standard **ListBox** control and the right has two **C1ComboBoxe**s. These are the controls we will populate and style in the following steps.

### *Populate the Controls*

Before we start using templates and styles, let us populate the controls first. To do that, complete the following:

1. Open the **MainPage.xaml.cs** file and paste the following code into the page constructor:

```
public Page()
{
    InitializeComponent();

    // Get list of items
    IEnumerable list = GetItems();

    // Add items to ListBox and in C1ComboBox
```

```
    _listBox.ItemsSource = list;
    _cmb1.ItemsSource = list;

    // Show fonts in the other C1ComboBox
    FontFamily[] ff = new FontFamily[]
    {
       new FontFamily("Default font"),
       new FontFamily("Arial"),
       new FontFamily("Courier New"),
       new FontFamily("Times New Roman"),
       new FontFamily("Trebuchet MS"),
       new FontFamily("Verdana")
    };
    _cmb2.ItemsSource = ff;
}
```

The code populates the **ListBox** and both **C1ComboBoxes** by setting their **ItemsSource** property. **ItemsSource** is a standard property present in most controls that support lists of items (**ListBox**, **DataGrid**, **C1ComboBox**, and so on).

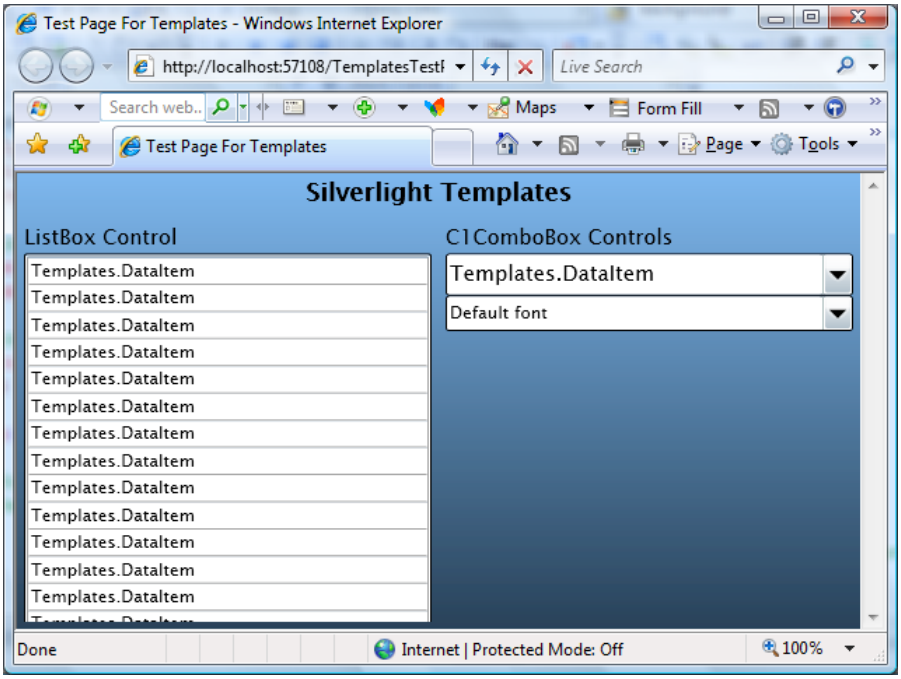
2. Add the following code to implement the **GetItems()** method in the MainPage class:

```
List<DataItem> GetItems()
{
  List<DataItem> members = new List<DataItem>();
  foreach (MemberInfo mi in this.GetType().GetMembers())
  {
    members.Add(new DataItem(mi));
  }
  return members;
}
```

3. Add the definition of the **DataItem** class. to the **MainPage.xaml.cs** file, below the MainPage class definition:

```
public class DataItem
{
    public string ItemName { get; set; }
    public MemberTypes ItemType { get; set; }
    public DataItem(MemberInfo mi)
    {
        ItemName = mi.Name;
        ItemType = mi.MemberType;
    }
}
```

If you run the project now, you will see that the controls are being populated. However, they don't do a very good job of showing the items:




The controls simply convert the **DataItem** objects into strings using their **ToString()** method, which we didn't override and by default returns a string representation of the object type ("Templates.DataItem").

The bottom **C1ComboBox** displays the font family names correctly. That's because the **FontFamily** class implements the **ToString()** method and returns the font family name.

It is easy to provide a **ToString()** implementation that would return a more useful string, containing one or more properties. For example:

```
public override string ToString()
{
    return string.Format("{0} {1}", ItemType, ItemName);
}
```

If you add this method to the **DataItem** class and run the project again, you will see a slightly more satisfying result. But there's only so much you can do with plain strings. To represent complex objects effectively, we need something more. Enter Data Templates!

### *Defining and Using Data Templates*

Data Templates are objects that map regular .NET objects into **UIElement** objects. They are used by controls that contain lists of regular .NET objects to convert these objects into **UIElement** objects that can be displayed to the user.

For example, the Data Template below can be used to map our **DataItem** objects into a **StackPanel** with two **TextBlock** elements that display the **ItemName** and **ItemType** properties of the **DataItem**. This is what the template definition looks like in XAML markup:

```
<UserControl x:Class="Templates.MainPage"
  xmlns="http://schemas.microsoft.com/client/2007"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:C1_Silverlight="clr-
namespace:C1.Silverlight;assembly=C1.Silverlight">

  <!-- Data template used to convert DataItem objects into UIElement
objects -->
```

```
  <UserControl.Resources>
    <DataTemplate x:Key="DataItemTemplate" >
      <StackPanel Orientation="Horizontal" Height="30" >
        <TextBlock Text="{Binding ItemType}"
          Margin="5" VerticalAlignment="Bottom" Foreground="Red"
FontSize="10" />
        <TextBlock Text="{Binding ItemName}"
          Margin="5" VerticalAlignment="Bottom" />
      </StackPanel>
    </DataTemplate>
  </UserControl.Resources>

  <!-- Page content (same as before)... -->
```

This template tells Silverlight (or WPF) that in order to represent a source data object, it should do this:

1. Create a **StackPanel** with two **TextBlock**s in it,
2. Bind the **Text** property of the first **TextBlock** to the **ItemType** property of the source data object, and
3. Bind the **Text** property of the second **TextBlock** object to the **ItemName** property of the source object.

That's it. The template does not specify what type of control can use it (any control can, we will use it with the **ListBox** and also with the **C1ComboBox**), and it does not specify the type of object it should expect (any object will do, as long as it has public properties named **ItemType** and **ItemName**).

To use the template, add an **ItemTemplate** attribute to the controls where you want the template to be applied. In our example, we will apply it to the **ListBox** declaration in the **MainPage.xaml** file:

```
<!-- ListBox on the left -->
<StackPanel Grid.Row="1" Margin="5" >
  <TextBlock Text="ListBox Control" />
  <ListBox x:Name="_listBox"
           ItemTemplate="{StaticResource DataItemTemplate}" />
</StackPanel>
```

And also to the top **C1ComboBox**:

```
<!-- C1ComboBox on the right -->
<StackPanel Grid.Column="2" Grid.Row="1" Margin="5" >
  <TextBlock Text="C1ComboBox Controls" />

  <!-- C1ComboBox 1 -->
  <C1_Silverlight:C1ComboBox x:Name="_cmb1" Margin="0,0,0,5"
        ItemTemplate="{StaticResource DataItemTemplate}" />
```

Note that we can now change the appearance of the **DataItem** objects by modifying the template in one place. Any changes will automatically be applied to all objects that use that template, making application maintenance much easier.

Before you run the application again, let's add a template to the second **C1ComboBox** as well. This control contains a list of font families. We can use templates to display each item using the actual font they represent.

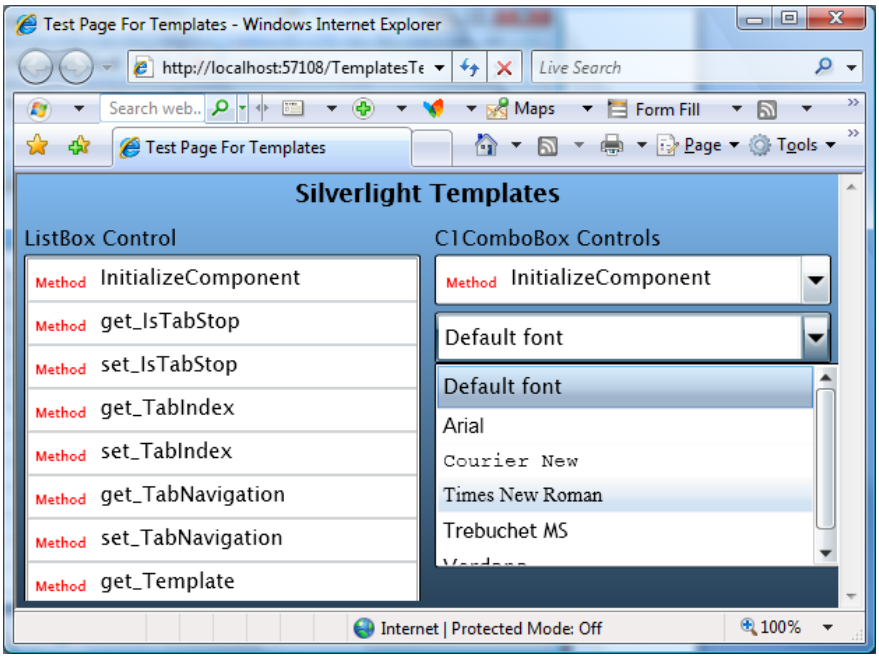
This time, we will not define the template as a resource. It will only be used in one place, so we can insert it inline, as shown below:

```
<!-- C1ComboBox 2 -->
<C1_Silverlight:C1ComboBox x:Name="_cmb2" FontSize="12" Margin="0,0,0,5" >
  <C1_Silverlight:C1ComboBox.ItemTemplate>
    <DataTemplate>
      <TextBlock Text="{Binding}" FontFamily="{Binding}" Margin="4" />
    </DataTemplate>
  </C1_Silverlight:C1ComboBox.ItemTemplate>
</C1_Silverlight:C1ComboBox>
```

Don't let the XAML syntax confuse you. This specifies that in order to create items from data, the control should use a **DataTemplate** that consists of a single **TextBlock** element. The **TextBlock** element should have two of its properties (**Text** and **FontFamily**) bound to the data object itself (as opposed to properties of that object).

In this case, the data object is a **FontFamily** object. Because the template assigns this object to the **Text** property and also to the **FontFamily** property, the **TextBlock** will display the font name and will use the actual font.

If you run the project now, you should see this result:




Note that if you assign a **DataTemplate** to the **C1ComboBox**, it will no longer be able to perform text-related tasks such as auto-search and editing. If you want to re-enable those features, you should provide your own **ItemConverter** that is a standard **TypeConverter**.

Styles and Templates are extremely powerful concepts. We encourage you to play and experiment with this sample. Try modifying the templates to show the data in different ways. The more you experiment, the more comfortable you will feel with these concepts and with the Silverlight/WPF application architecture.

## Control Templates

Data Templates allow you to specify how to convert arbitrary data objects into **UIElement** objects that can be displayed to the user. But that's not the only use of templates in Silverlight and WPF. You can also use templates to modify the visual structure of existing **UIElement** objects such as controls.

Most controls have their visual appearance defined by a native XAML resource (typically contained within the assembly that defines the control). This resource specifies a **Style** which assigns values to most of the control's properties, including its **Template** property (which defines the control's internal "visual tree").

For example:

```
<Style TargetType="HyperlinkButton">
  <Setter Property="IsEnabled" Value="true" />
  <Setter Property="IsTabStop" Value="true" />
  <Setter Property="Foreground" Value="#FF417DA5" />
  <Setter Property="Cursor" Value="Hand" />
  <Setter Property="Template">
```

```
    <Setter.Value>
      <ControlTemplate TargetType="HyperlinkButton">
        <Grid x:Name="RootElement" Cursor="{TemplateBinding Cursor}">
          <!-- Focus indicator -->
          <Rectangle x:Name="FocusVisualElement" StrokeDashCap="Round"
...=""/>
          <!-- HyperlinkButton content -->
          <ContentPresenter x:Name="Normal"
               Background="{TemplateBinding Background}"
               Content="{TemplateBinding Content}"
               ContentTemplate="{TemplateBinding ContentTemplate}"...=""/>
        </Grid>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

This is a very simplified version of the XAML resource used to specify the **HyperlinkButton** control. It consists of a **Style** that begins by setting the default value of several simple properties, and then assigns a value of type **ControlTemplate** to the control's **Template** property.

The **ControlTemplate** in this case consists of a **Grid** (*RootElement*) that contains a **Rectangle** (*FocusVisualElement*) used to indicate the focused state and a **ContentPresenter** (*Normal*) that represents the content portion of the control (and itself contains another **ContentTemplate** property).

Note the *TemplateBinding* attributes in the XAML. These constructs are used to map properties exposed by the control to properties of the template elements. For example, the **Background** property of the hyperlink control is mapped to the **Background** property of the *Normal* element specified in the template.

Specifying controls this way has some advantages. The complete visual appearance is defined in XAML and can be modified by a professional designer using Expression Blend, without touching the code behind it. In practice, this is not as easy as it sounds, because there are logical relationships between the template and the control implementation.

Recognizing this problem, Silverlight introduced a **TemplatePart** attribute that allows control classes to specify the names and types it expects its templates to contain. In the future, this attribute will be added to WPF as well, and used by designer applications such as Blend to validate templates and ensure they are valid for the target control.

For example, the Microsoft **Button** control contains the following **TemplatePart** attributes:

```
/// <summary>
/// Represents a button control, which reacts to the Click event.
/// </summary>
[TemplatePart(Name = Button.ElementRootName, Type =
typeof(FrameworkElement))]
[TemplatePart(Name = Button.ElementFocusVisualName, Type =
typeof(UIElement))]
[TemplatePart(Name = Button.StateNormalName, Type = typeof(Storyboard))]
[TemplatePart(Name = Button.StateMouseOverName, Type =
typeof(Storyboard))]
[TemplatePart(Name = Button.StatePressedName, Type = typeof(Storyboard))]
[TemplatePart(Name = Button.StateDisabledName, Type = typeof(Storyboard))]
public partial class Button : ButtonBase
```

These six template parts constitute a contract between the control implementation and the design specification. They tell the designer that the control implementation expects to find certain elements in the template (defined by their name and type).

Well-behaved controls should degrade gracefully, not crashing if some non-essential elements are missing from the template. For example, if the control can't find a **Storyboard** named *Button.StateMouseOverName* in the template, it should not do anything when the mouse hovers over it.

Well-implemented templates should fulfill the contract and provide all the elements that the control logic supports. Designer applications such as Blend can enforce the contract and warn designers if they try to apply invalid templates to controls.

For the time being, the easiest way to create new templates for existing controls is to start with the original XAML and customize it.

We will not show any actual examples of how to create and use custom control templates here. Instead, we suggest you download the examples developed by Corrina Barber:

http://blogs.msdn.com/corrinab/archive/2008/03/11/silverlight-2-control-skins.aspx

The link contains previews and downloads for three 'skins' (bubbly, red, and flat). Each skin consists of a set of **Style** specifications, similar to the one shown above, which are added to the application's global XAML file (App.xaml). The format is similar to this:

```
<Application xmlns="http://schemas.microsoft.com/client/2007"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="Styles_Red.App"

<Application.Resources>
  <!-- Button -->
  <Style x:Key="buttonStyle" TargetType="Button">
    <Setter Property="IsEnabled" Value="true" />
    <Setter Property="IsTabStop" Value="true" />
    <Setter Property="Foreground" Value="#FF1E2B33" />
    <Setter Property="Cursor" Value="Hand" />
    <Setter Property="TextAlignment" Value="Center" />
    <!-- A lot more XAML follows… -->
```

Once these styles are defined in the **App.xaml** file, they can be assigned to any controls in the application:

```
<Button Content="Button" Style="{StaticResource buttonStyle}"/>
```

If you are curious, this is what the **Button** control looks like after applying each of the skins defined in the reference above:

Default


Bubbly


Red


Flat


This mechanism is extremely powerful. You can change what the controls look like and even the parts used internally to build them.

Unlike data templates, however, control templates are not simple to create and modify. Creating or changing a control template requires not only design talent but also some understanding of how the control works.

It is also a labor-intensive proposition. In addition to their normal appearance, most controls have **Storyboards** that are applied to change their appearance when the mouse hovers over them, when they gain focus, get pressed, get disabled, and so on (see the **C1ComboBox** example above).

Furthermore, all controls in an application should appear consistent. You probably wouldn't want to mix bubbly buttons with regular scrollbars on the same page for example. So each 'skin' will contain styles for many controls.

Some controls are designed with custom templates in mind. For example, the **C1ComboBox** has an **ItemsPanel** property of type **ItemsPanelTemplate**. You can use this property to replace the default drop-down **ListBox** element with any other **UIElement** you like.

For examples of using the **ItemsPanel** property, check the **ControlExplorer** sample installed by default with **ComponentOne Studio for Silverlight**.

## Preparing Your Enterprise Environment

Several considerations are important to take into account when planning a corporate deployment of your Silverlight applications in an enterprise environment. For information about these considerations and a description of system requirements and deployment methods as well as the techniques to maintain and support Silverlight after deployment, please see the Silverlight Enterprise Deployment Guide provided by the Microsoft Silverlight team.

The guide helps you to plan and carry out a corporate deployment of Silverlight, and covers:

- Planning the deployment
- Testing deployment strategy
- Deploying Silverlight
- Maintaining Silverlight in your environment

The Silverlight Enterprise Deployment Guide is available for download from the Silverlight whitepapers site: http://silverlight.net/learn/whitepapers.aspx.

# Key Features

**ComponentOne Chart for Silverlight** includes several key features, such as:

- **30+ Chart Types**

  Get the exact chart representation you need for your Silverlight application by choosing from over 30 popular 2D chart types. Available chart types include line, scatter, bar, column, pie, area and many more.

- **Professionally Designed with Just Two Properties**

  Chart includes 12 built-in themes and 22 built-in palettes. Both are set with just one property inside Visual Studio! Themes apply to the entire chart area while palettes apply to just the chart elements (bars, points, pie slices, etc). Combine themes with different palettes to effortlessly make an endless combination of looks.

- **Labels and Tooltips**

  Display relative data values over chart elements as labels or tooltips. Any UI elements can be used as labels and tooltips allowing full customization.

- **Chart Legends**

  Create a separate chart legend using the C1ChartLegend control that connects to the chart through one property. This design provides maximum flexibility when styling and positioning the legend.

- **Fully Interactive Charts**

  Enhance the end-user experience by allowing end users to zoom in and out, scale, and scroll the chart.

- **Multiple Axes**

  Chart supports multiple, dependent axes that are added to the chart by simply defining an Axis object and adding it to the chart's View.Axes collection.

- **Logarithmic Axis Scales**

  Chart supports logarithmic axis scales of any base.

- **Trend Lines**

  Analyze your charted data with trend lines. Chart supports several automatic trend lines including polynomial, exponential, logarithmic, power, fourier, averages, minimums and maximums.

- **Lighting and Shadow Effects**

Create borders with lighting effects and add soft or hard shadows behind plot elements.

- **Stacked Charts**

  Stacking charts offers a simplified approach for representing complex data. Line, area, bar, radar, and plot charts can be stacked to display more complex data in a smaller space.

- **Dynamic Graphics**

  Chart takes advantage of the dynamic graphics available in the Silverlight platform including perspective and animation.

- **Silverlight Toolkit Themes Support**

  In addition to the 12 built-in themes, Chart ships with the most popular Microsoft Silverlight Toolkit themes including ExpessionDark, ExpressionLight, WhistlerBlue, RainerOrange, ShinyBlue, and BureauBlack.

# Chart for Silverlight Quick Start

The following quick start guide is intended to get you up and running with **Chart for Silverlight**. In this quick start, you'll start in Visual Studio to create a new project with a C1Chart control.

## Step 1 of 4: Creating an Application with a C1Chart Control

In this step, you'll begin in Visual Studio to create a Silverlight application using **Chart for Silverlight**.

Complete the following steps:

4. In Visual Studio 2008, select **File | New | Project**.
5. In the **New Project** dialog box, select a language in the left pane, and in the templates list select **Silverlight Application**. Enter a **Name** for your project and click **OK**. The **New Silverlight Application** dialog box will appear.
6. Click **OK** to close the **New Silverlight Application** dialog box and create your project.
7. In the XAML window of the project, resize the **UserControl** by changing `DesignWidth="640" DesignHeight="480"` to `DesignWidth="Auto" DesignHeight="Auto"` in the `<UserControl>` tag so that it appears similar to the following:

```
<UserControl x:Class="ChartSilverlight_Quick_Start.MainPage"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d" d:DesignWidth="Auto" d:DesignHeight="Auto">
```

   The **UserControl** will now resize to accommodate any content placed within it.
8. In the XAML window of the project, place the cursor between the `<Grid>` and `</Grid>` tags and click once. Note that you cannot currently add Silverlight controls directly to the design area in Visual Studio, so you must add them to the XAML window as directed in the next step.
9. Navigate to the Toolbox and double-click the C1Chart icon to add the control to the grid. The XAML markup resembles the following:

```
<Grid x:Name="LayoutRoot">
<c1chart:C1Chart></c1chart:C1Chart>
</Grid>
```

10. Inside the Grid, initialize the **C1Chart** control and give it a name by adding `x:Name="c1Chart1"` to the `<c1chart:C1Chart>` tag so that it appears similar to the following:

```
<c1chart:C1Chart Margin="0,0,8,8" MinHeight="160" MinWidth="240"
Name="c1Chart1">
</c1chart:C1Chart>
```

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

**What You've Accomplished**

You've successfully created a Silverlight application containing a C1Chart control. In the next step, you will Step 2 of 4: Adding Data to the Chart (page 19), you will add the data for C1Chart.

## 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 programmatically in the code behind file**

11. Right-click on the MainPage.xaml file and select **View Code** to open the editor.
12. Add the C1.Silverlight.C1Chart namespace directive

- Visual Basic

```
Imports C1.Silverlight.Chart
```

- C#

```
using C1.Silverlight.Chart;
```

13. 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#

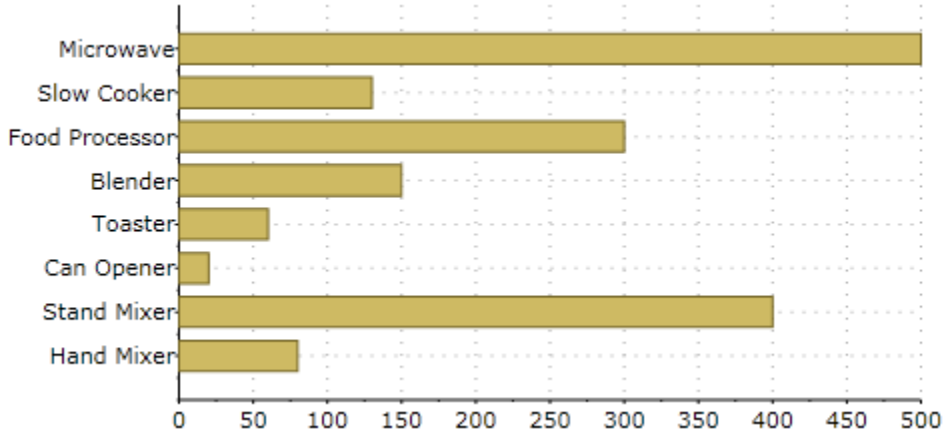
```
// Clear previous data
   c1Chart1.Data.Children.Clear();

   // 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 21), you'll learn how to customize the axes programmatically

**What You've Accomplished**

You have successfully added data to **C1Chart** so when you run your application the string values appear on the Y-axis like the following:




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

# 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 Silverlight programmatically in the code behind file**

Add the following code in the **Public MainPage()** 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
    c1Chart1.View.AxisX.MajorUnit = 20

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

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

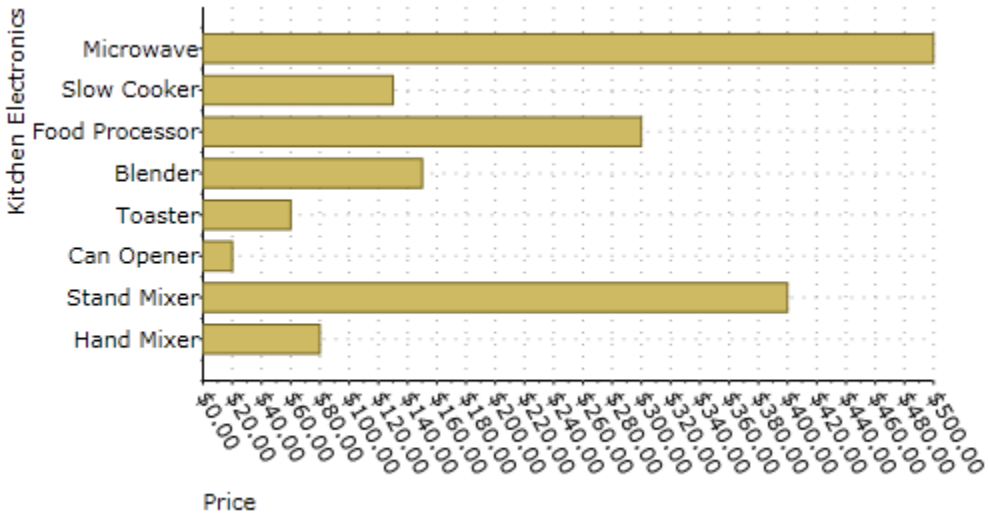
- C#

```
// 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;
    c1Chart1.View.AxisX.MajorUnit = 20;

    // 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 22), you'll learn how to customize the chart's appearance programmatically.

**What You've Accomplished**

You have successfully formatted the Chart Axes so when you run your application the new format for the axis annotation is applied to the chart.




In the next step you will apply a theme and palette to C1Chart to adjust its appearance.

# Step 4 of 4: Adjust the Chart's Appearance

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

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

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

- Visual Basic

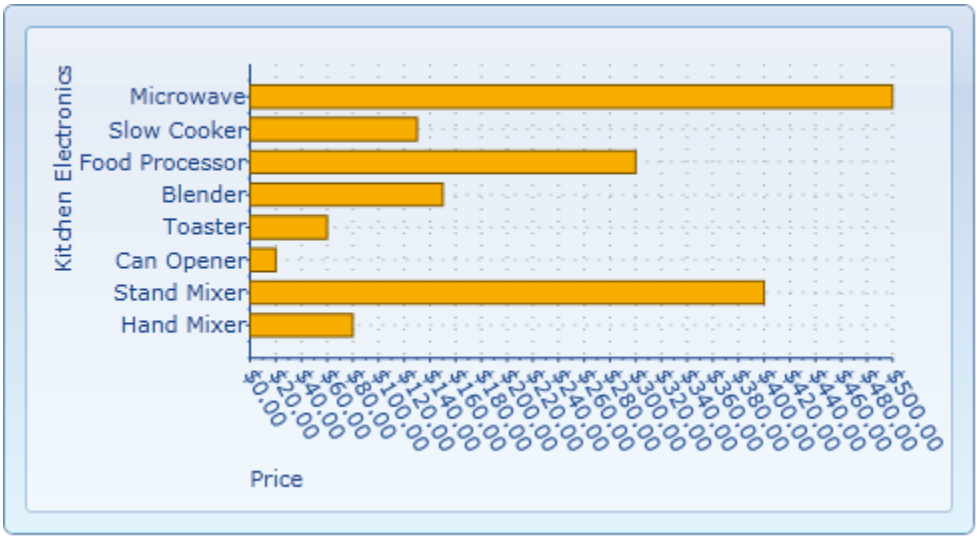
```
c1Chart1.Palette = Palette.Module;
            c1Chart1.Theme = ChartTheme.Office2007Blue;
```

- C#

```
c1Chart1.Palette = Palette.Module;
c1Chart1.Theme = ChartTheme.Office2007Blue;
```

**What You've Accomplished**

You successfully applied the **Office 2007 Blue** theme and the **Module** to the **C1Chart** control. When you run your application it appears like the following:




Congratulations! You've completed the **Chart for Silverlight** 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

The following XAML shows how to choose the chart type, choose the palette, set up axes, and add a data series:

```
        <c1:C1Chart x:Name="_chart" Palette="Module" ChartType="Line"
Foreground="#a0000000" Background="#e0ffffff" >

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

```
                    </c1:ChartView.AxisY>
                </c1:ChartView>
            </c1:C1Chart.View>

            <c1:C1ChartLegend Position="Right" />

            <c1:C1Chart.Data>
                <c1:ChartData ItemNames="P1 P2 P3 P4 P5">
                    <c1:DataSeries Label="s1" Values="20, 22, 19, 24,
25"
ConnectionStrokeThickness="6" />
                    <c1:DataSeries Label="s2" Values="8, 12, 10, 12,
15" />
                </c1:ChartData>
            </c1:C1Chart.Data>
        </c1:C1Chart>
```

# Chart for Silverlight Top Tips

The following top tips for Chart for Silverlight 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
```

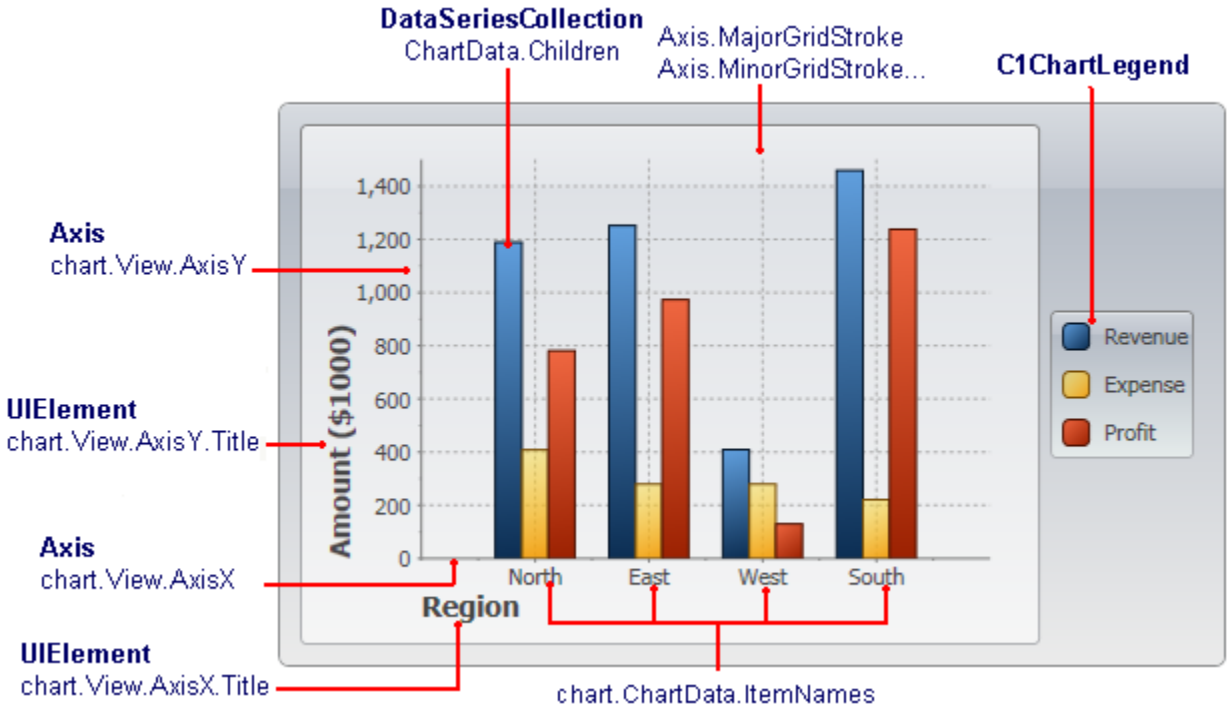
- C#

```
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:

14. 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.

15. 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.
16. 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.
17. 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.

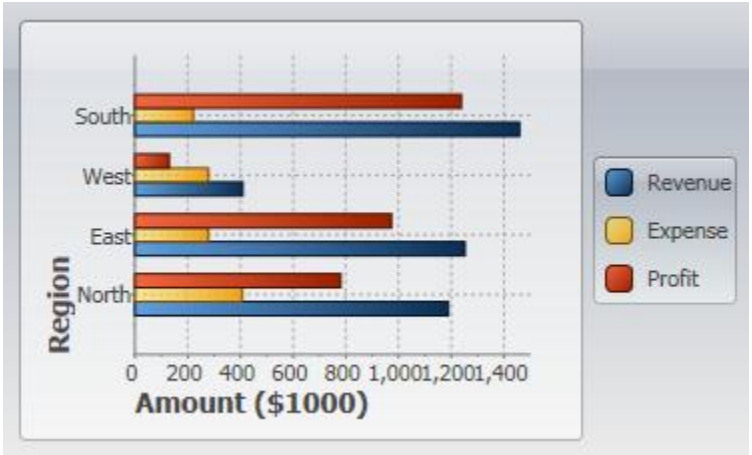
# Common Usage for Basic 2D Charts

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 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:

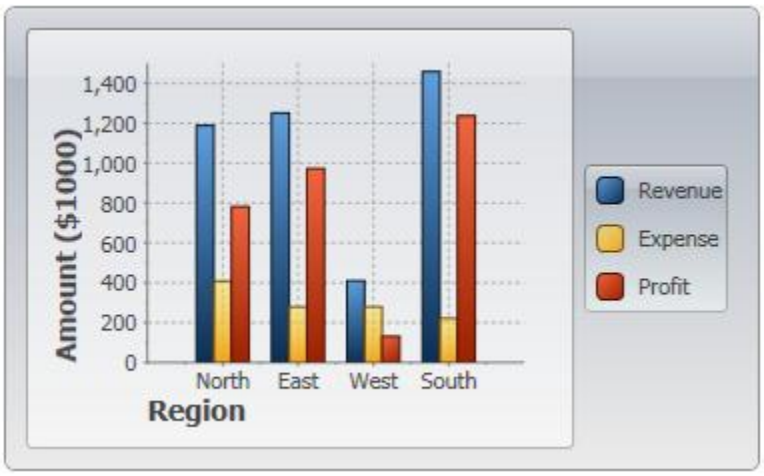
```
// Set theme
c1Chart.Theme = _c1Chart.TryFindResource(new
ComponentResourceKey(typeof(C1.Silverlight.C1Chart.C1Chart),"Office2007
Black")) as ResourceDictionary;}
```

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

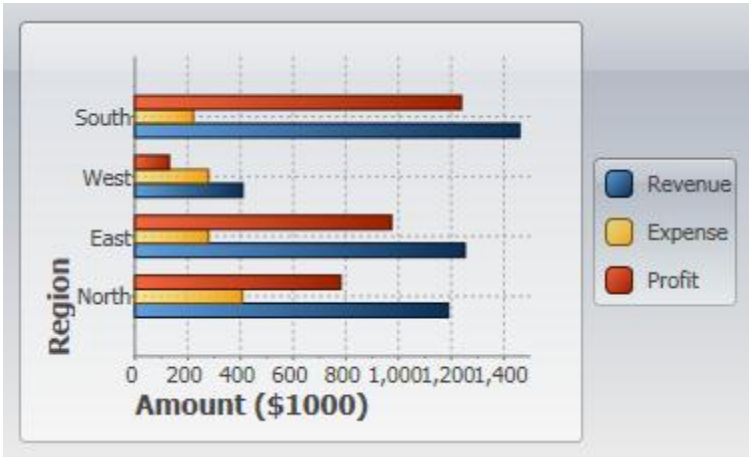
```
TextBlock CreateTextBlock(string text, double fontSize, FontWeight
fontWeight)
{
  var tb = new TextBlock();
  tb.Text = text;
  tb.FontSize = fontSize;
  tb.FontWeight = fontWeight;
  return tb;
}
```

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 simply 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:

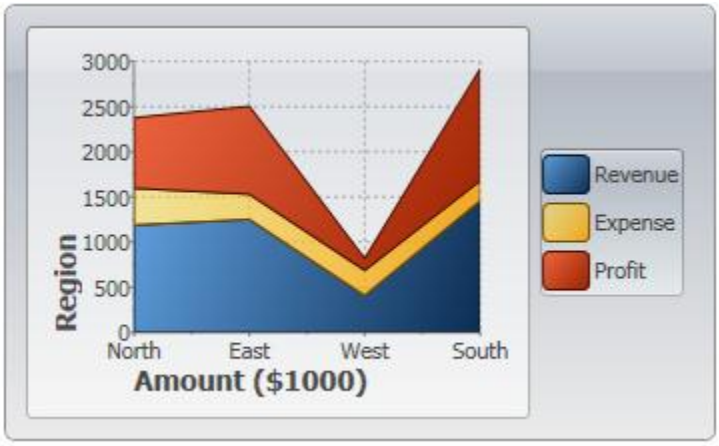
**ChartType.Column**




**ChartType.Bar**




**ChartType.AreaStacked**




**ChartType.Pie**


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

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

# 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. This 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.

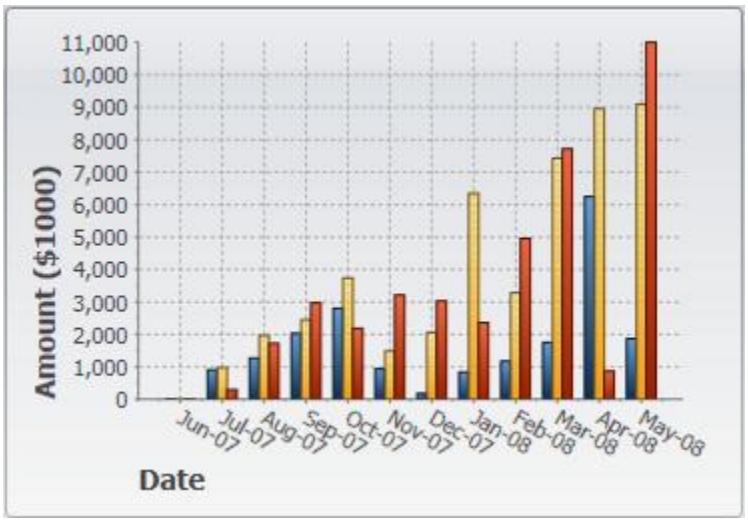
**Step 4:Adjust the chart's appearance**

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

```
c1Chart.Theme = "Office2007Black"
```

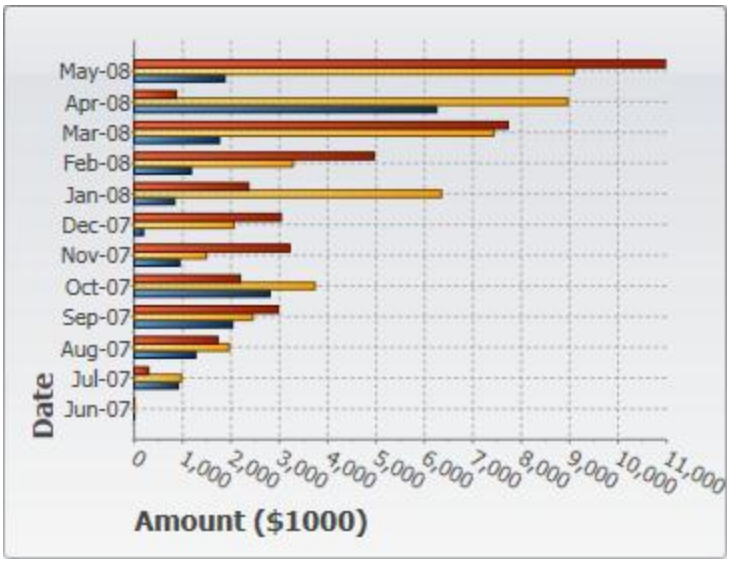
This concludes the code that generates our time-series charts. You can test it by running it and changing the ChartType property to Bar, Column, AreaStacked, or Pie to create charts of different types. The result should be similar to the images below:

**ChartType.Column**



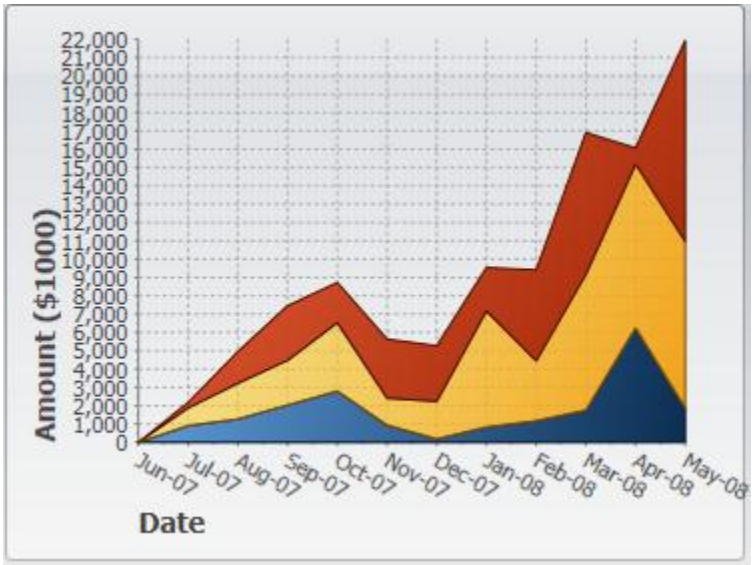

**Note:** The AnnoAngle property was set to "30" to make room for the Axis X labels in the images above.

**ChartType.Bar**



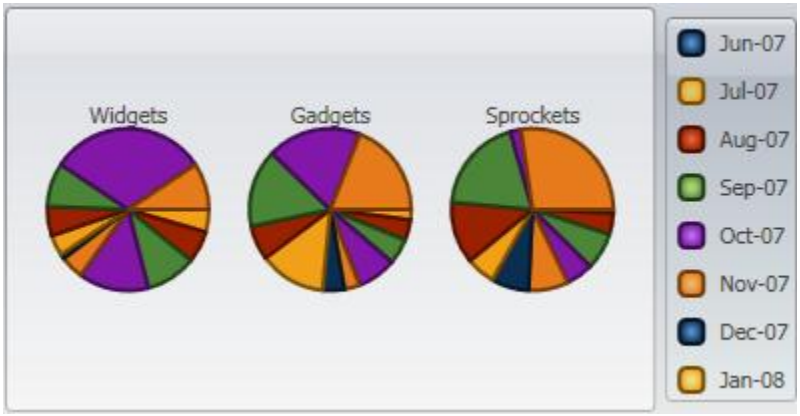
**Note:** The AnnoAngle property was set to "30" to make room for the Axis Y labels in the images above.

**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 also 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:

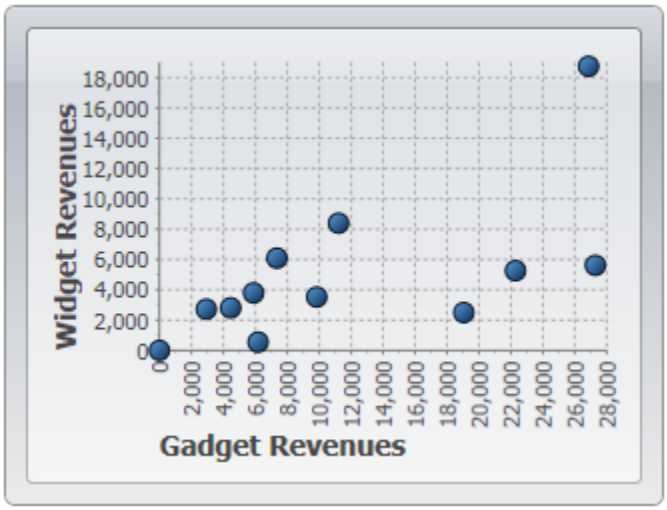
```
// create the new XYDataSeries
var ds = new XYDataSeries();

// 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:

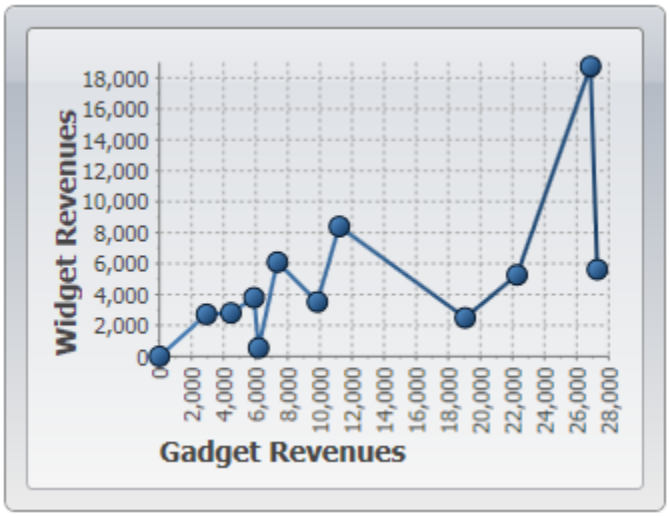
```
c1Chart.Theme = Theme.Office2007Black;
        }
```

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:

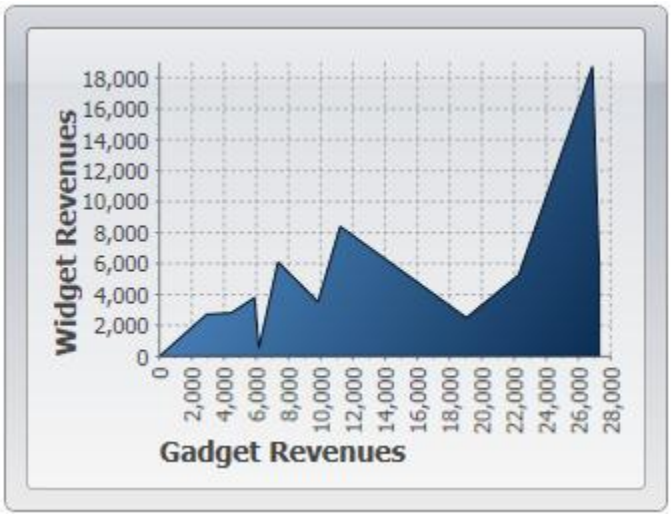
**ChartType.XYPlot**




**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:

18. **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 30) 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 76).
19. **Axis**: The chart samples used in the Common Usage for Basic 2D Charts (page 30) 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 73).

    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.
20. **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 79).
21. **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 77).

# Specialized Charts

There are a few specialized chart types that are slightly different from the ones described so far. These are described in the following sections.

## Financial Charts

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

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;
```

Note that the **SymbolSize.Width** property of the data series can be used to change the width of the chart symbols.

# Gantt Charts

**C1Chart** implements Gantt charts, which show tasks organized along time.

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.

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:

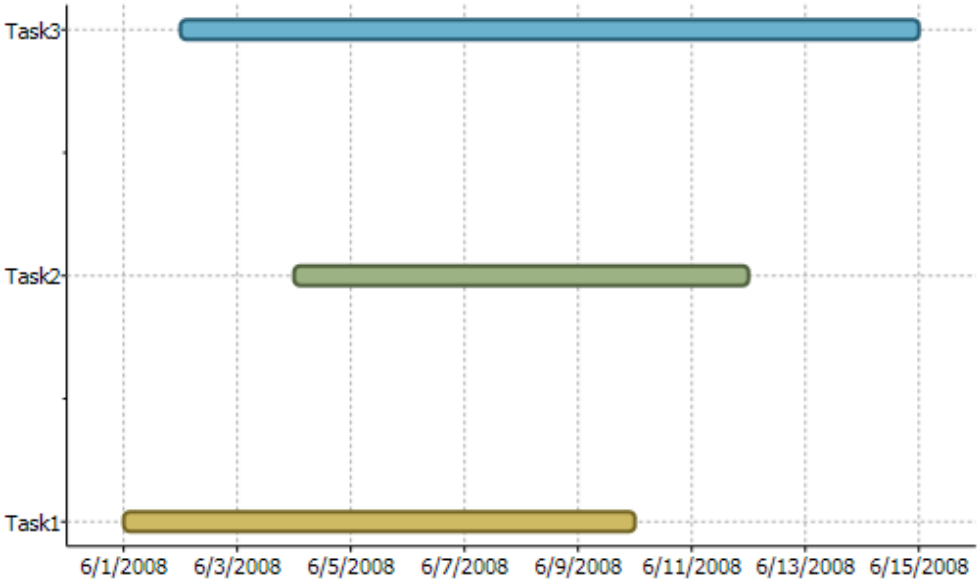
```
private void CreateGanttChart()
{
  // Clear current chart
  _c1Chart.Reset(true);

  // 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.

The final result looks like this:




# Chart Types

This section introduces all of the specific chart types available in C1Chart.

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 |

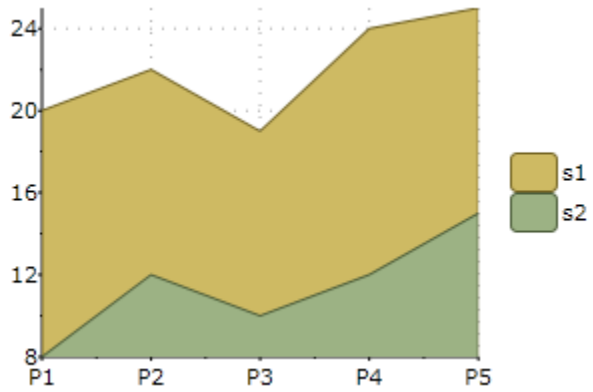
# Area Charts

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 Silverlight** supports the following types of Area charts:

- AreaSmoothed
- AreaStacked
- AreaStacked100pc

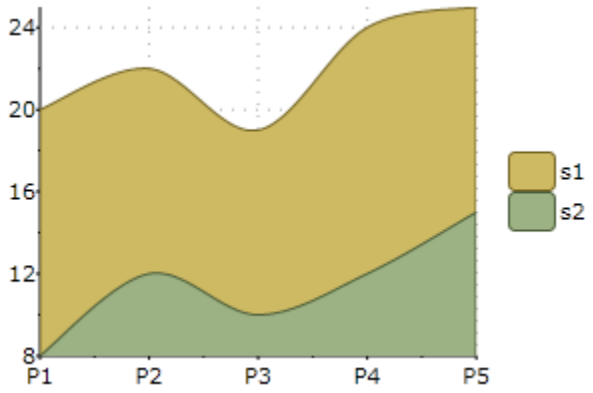
**Area**

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



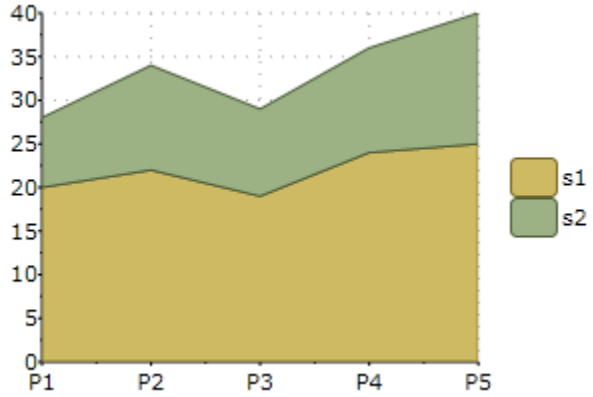

**Area Smoothed**

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



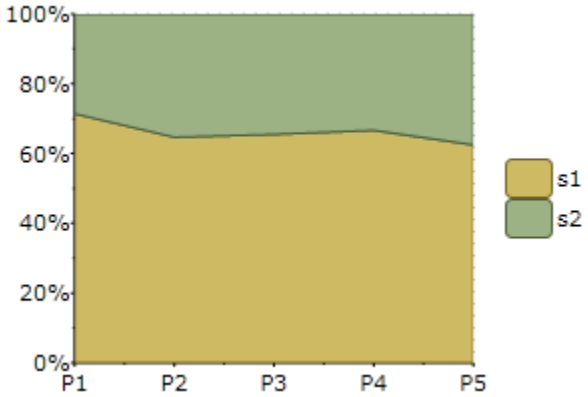

**Area Stacked**

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**:



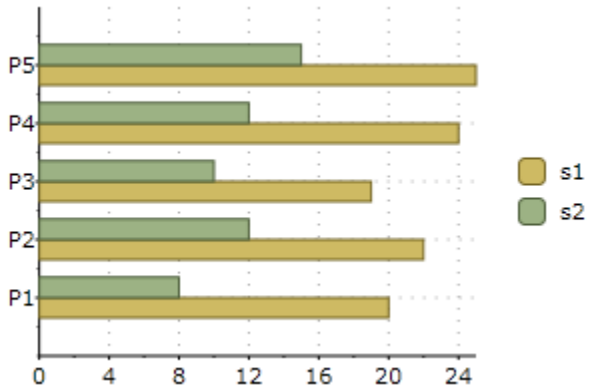

# Bar Charts

Chart for Silverlight supports the following types of Bar charts:

- Bar
- BarStacked
- BarStacked100pc

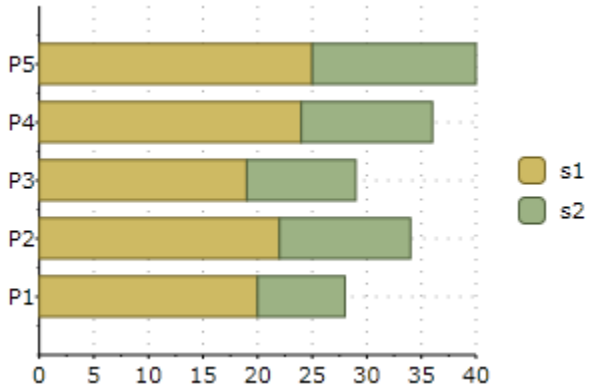
**Bar**

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



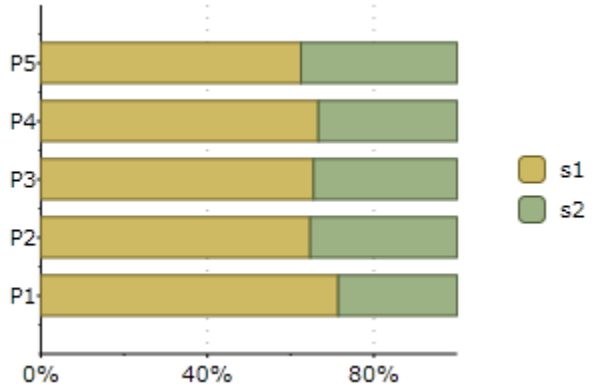

**Bar Stacked**

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



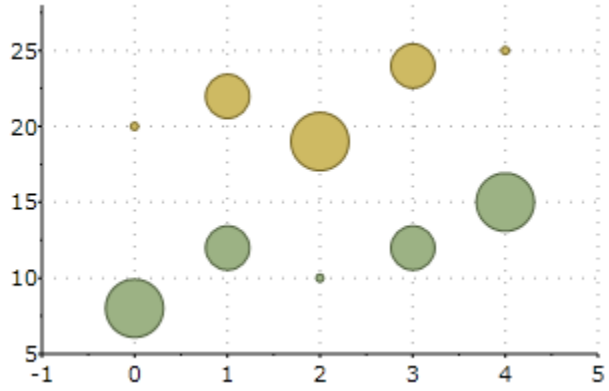

**Bar Stacked 100 Percent**

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 Height="200" Width="300" ChartType="Bubble"
Margin="56,45,284,235"       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.

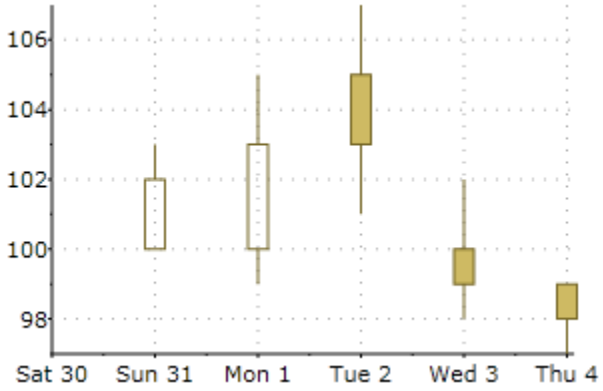
A **Candle** chart is a special type of **HiLoOpenClose** chart that is used to show the relationship between the open and close as well as the high and low. Like, **HiLoOpenClose** charts, **Candle** charts use the same price data (time, high, low, open, and close values) except they include a thick 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 XValuesSource, OpenValuesSource, CloseValuesSource, HighValuesSource, and LowValuesSource, like the following:

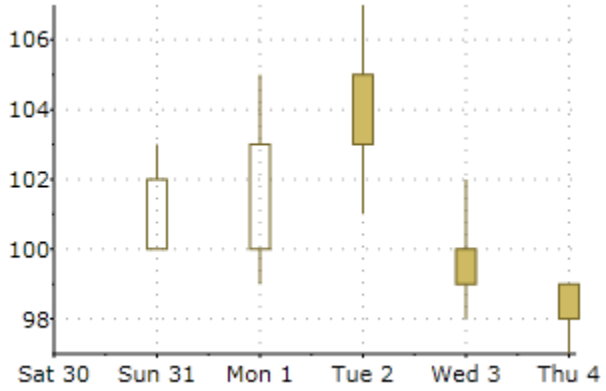
```
<c1chart:C1Chart ChartType="Candle">
    <c1chart:C1Chart.Data>
      <c1chart:ChartData>
        <c1chart:HighLowOpenCloseSeries
            XValues="1 2 3 4 5"
           HighValues="103 105 107 102 99"
           LowValues="100 99 101 98 97"
           OpenValues="100 100 105 100 99"
           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 XValuesSource, OpenValuesSource, CloseValuesSource, HighValuesSource, and LowValuesSource, 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;
```

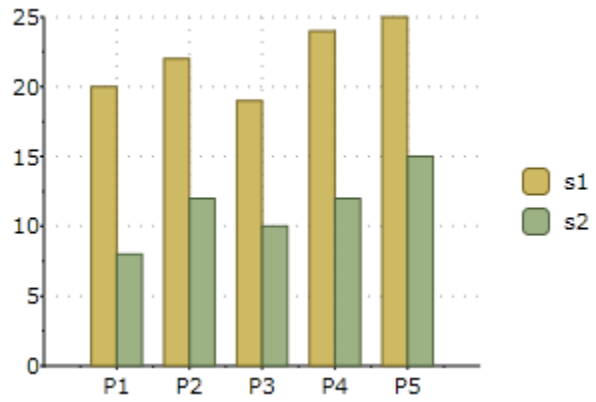
# Column Charts

Chart for Silverlight supports the following types of Column charts:

- Column
- ColumnStacked
- ColumnStacked100pc

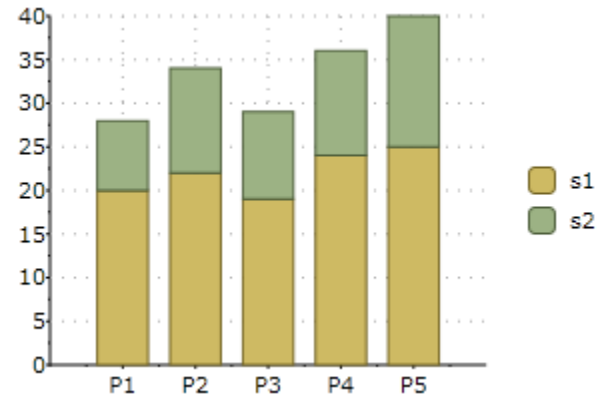
**Column**

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



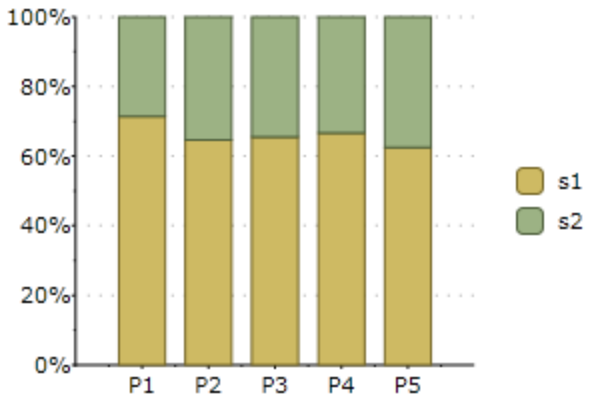

**Column Stacked**

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




**Column Stacked 100%**

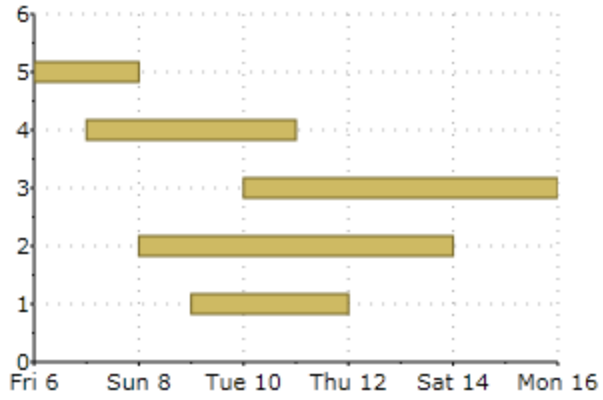
The following image represents the Column Stacked 100% when you set the ChartType property to **ColumnStacked100pc**:



# 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 when the following code is used:




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.

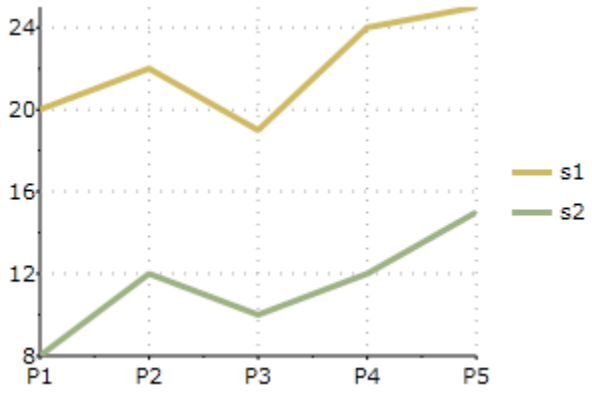
# Line Charts

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

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

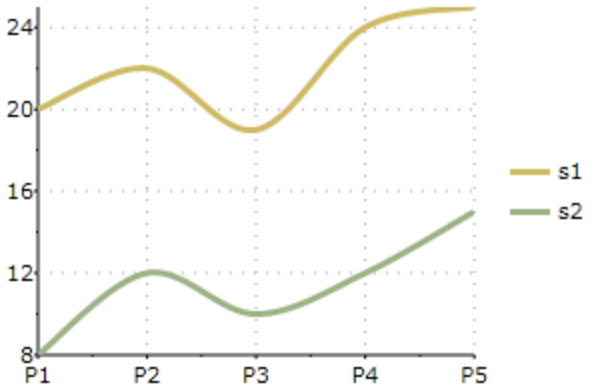
**Line**

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




**Line Smoothed**

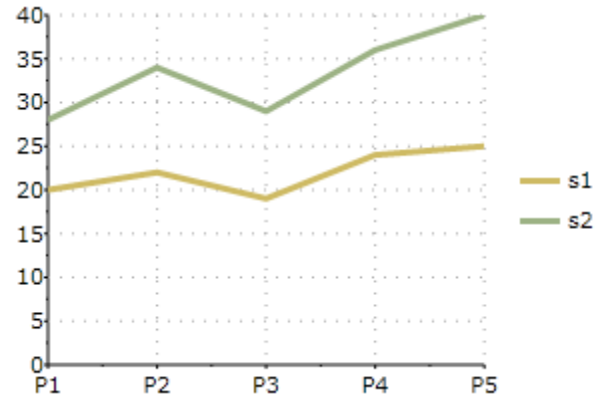
The following image represents the Line Smoothed chart when you set the ChartType property to **LineSmoothed**:




**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.

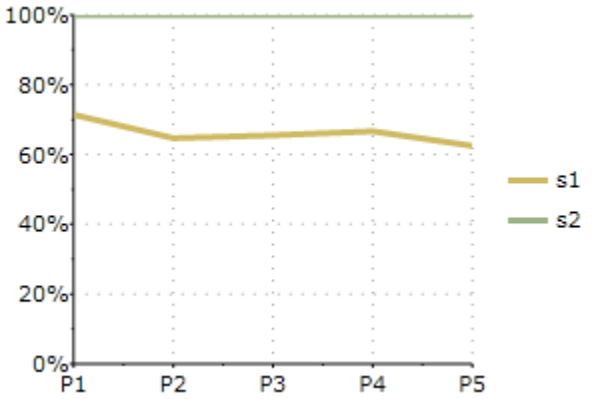
The following image represents the Line Stacked chart when you set the ChartType property to **LineStacked**:




**Line Stacked 100%**

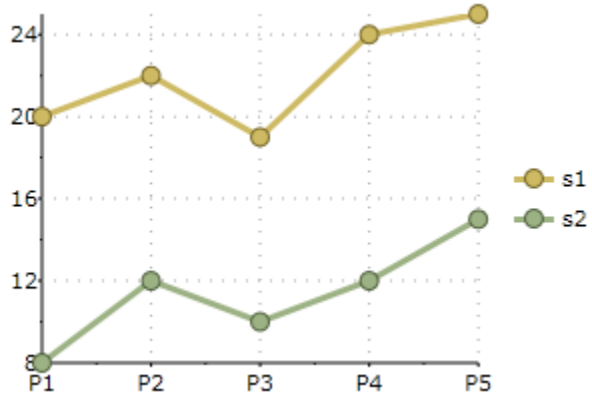
Select the **LineStacked100pc** member from the ChartType enumeration to create a specific stacking Line chart. Stacking charts represent the data by stacking the values for each series on top of the values from the previous series.

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



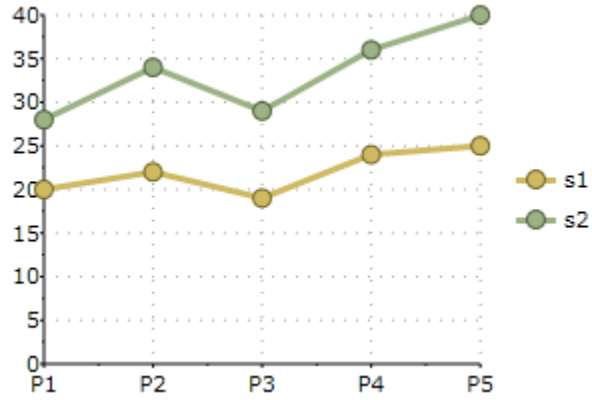

**Line Symbols**

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




**Line Symbols Stacked**

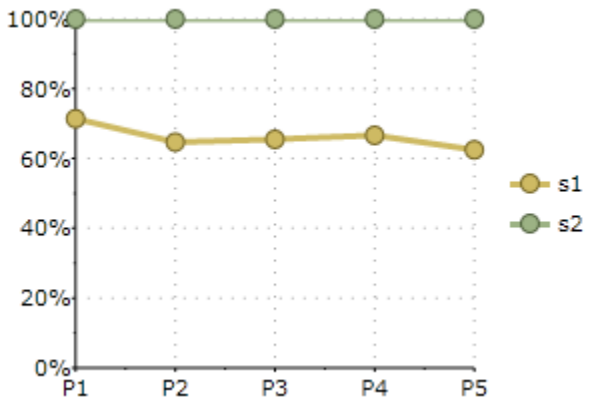
The following image represents the Line Symbols Stacked chart when you set the ChartType property to **LineSymbolsStacked**:



**Line Symbols Stacked 100%**

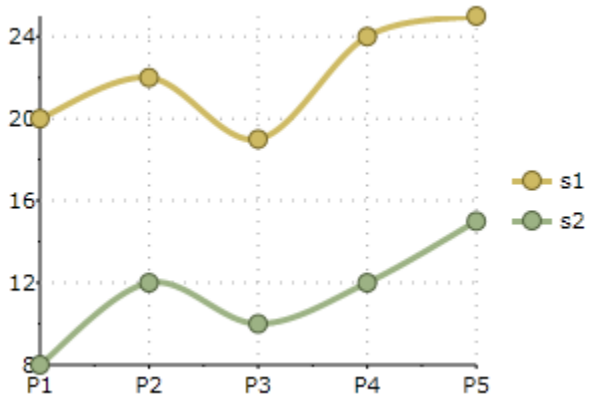
Select the **LineSymbolsStacked100pc** member from the ChartType enumeration to create a specific stacking Line chart. Stacking charts represent the data by stacking the values for each series on top of the values from the previous series.

The following image represents the Line Stacked 100% chart when you set the ChartType property to **LineSymbolsStacked100pc**:




**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.

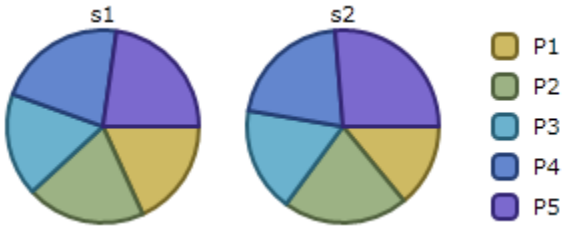
Pie charts have one significant difference when compared to other **C1Chart** chart types in Pie charts; each series represents one slice of the pie. Therefore, you will never have Pie charts with a single series (they would be just circles). In most cases, Pie charts have multiple series (one per slice)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

**Chart for Silverlight** supports the following types of Pie charts:

- Pie
- Dougnut Pie
- Exploded Pie
- Exploded Doughnut Pie

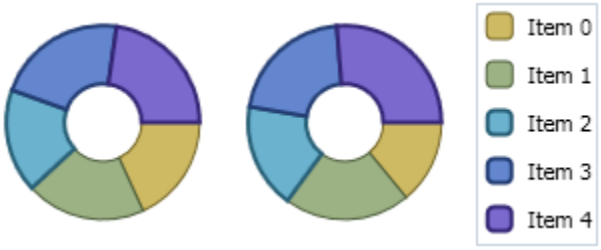
**Pie**

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



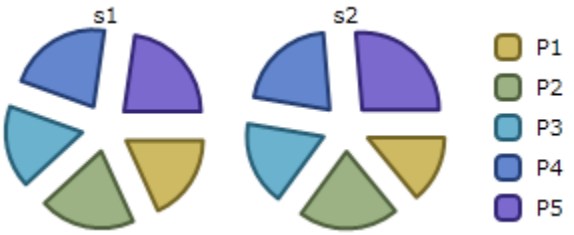

**Doughnut Pie**

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 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**

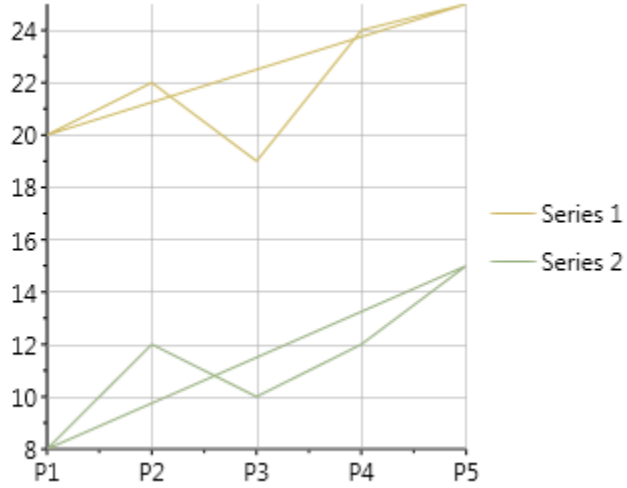
Use the **PieOptions.StartingAngleAttached** property to specify the angle at which the slices for the first series start. The default angle is 0 degrees. The angle represents the arc between the most clockwise edge of the first slice and the right horizontal radius of the pie, as measured in the counter-clockwise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

**Exploding Pies**

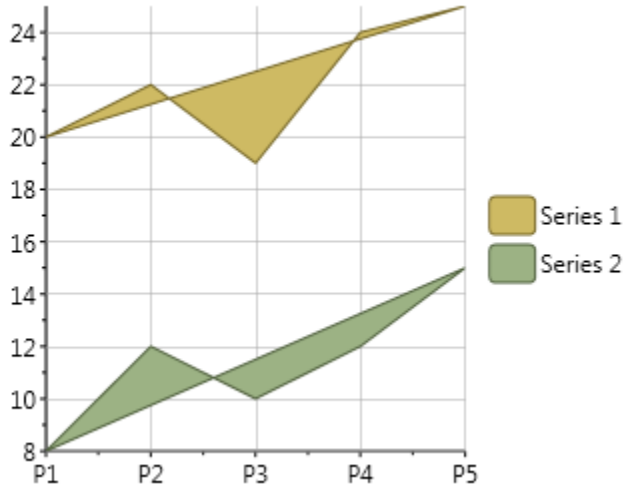
A slice of a Pie chart can be emphasized by exploding it, which extrudes the slice from the rest of the pie. Use the Offset property of the series to set the exploded slice's offset from the center of the pie. The offset is measured as a percentage of the radius of the pie.

## Polygon Charts

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**:




# Polar and Radar Charts

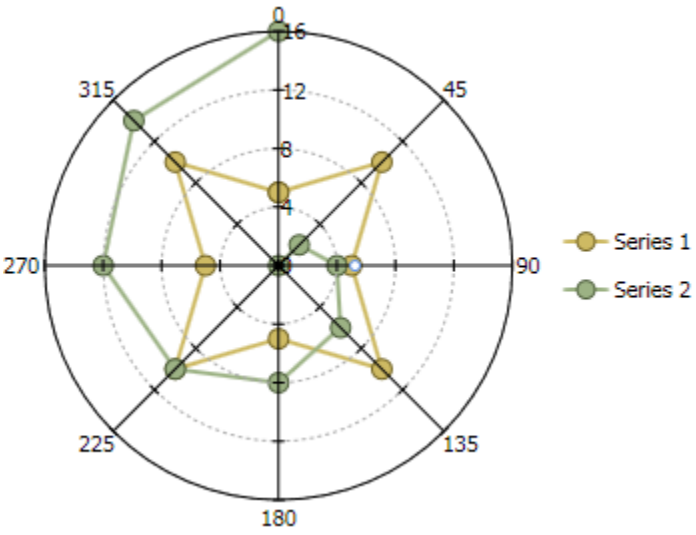
**Polar Charts**

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.

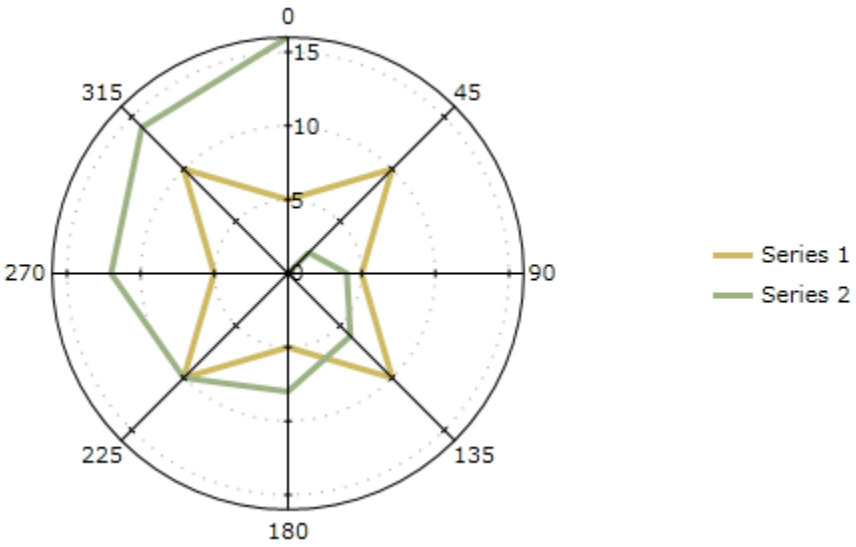
The following images represents the different types of Polar charts when you set ChartType property to **PolarLinesSymbol**, **PolarLines**, **PolarSymbols** and specifiy the data values for the XYDataSeries, like the following:

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



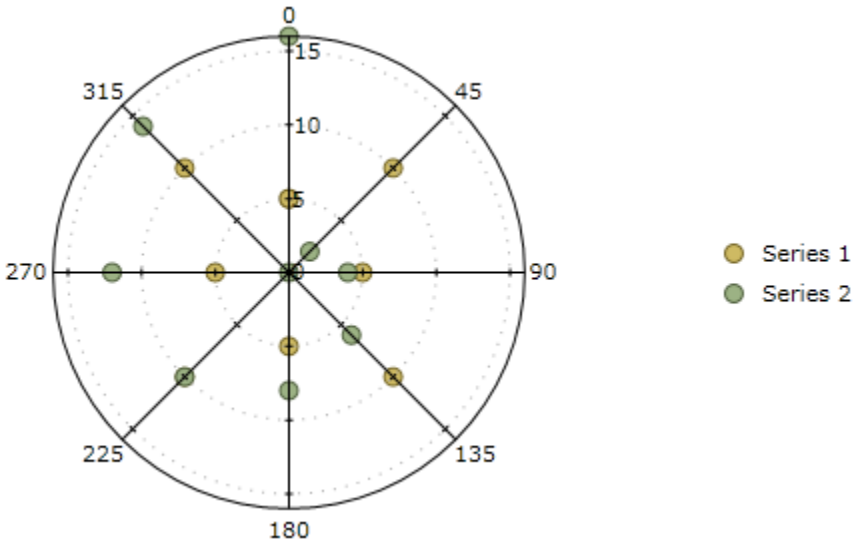

The following image represents the Polar chart with symbols and lines when you set ChartType property to **PolarLines**.

```
<c1chart:C1Chart Name="c1Chart1" ChartType="PolarLines">
  <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/> </c1chart:C1Chart>
```



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

```
<c1chart:C1Chart Name="c1Chart1" ChartType="PolarSymbols">
  <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/> </c1chart:C1Chart>
```

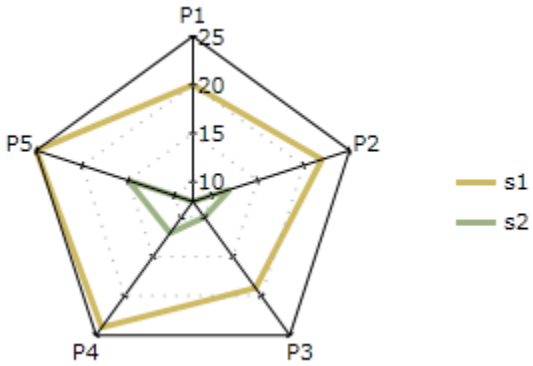



**Radar Charts**

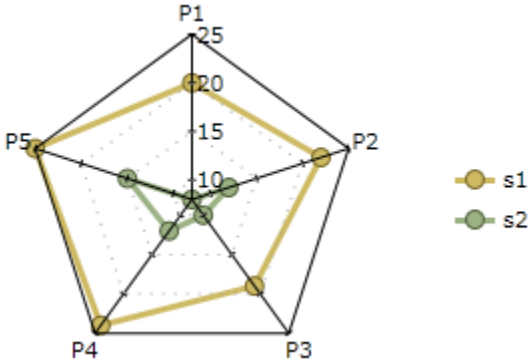
A Radar chart is a variation of a Polar chart. A Radar chart draws the y value in each data set along a radar line. If the data has n unique points, then the chart plane is divided into n equal angle segments, and a radar line is drawn (representing each point) at n/360 degree increments. By default,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

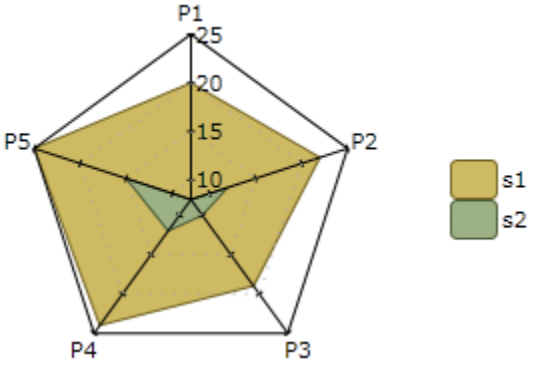
The following image represents the Radar chart when you set ChartType property to **Radar**.




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 has special properties to chart the degrees of the Radar, set the starting angle, create filled radar chart.and whether or not to use flat Y coordinate gridlines for radar charts.

### 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.

**Controlling Appearance of Labels on the Axes**

You can use the RadarLabelVisibility to control the appearance of the labels on the axes of radar chart. There are three available options that specify in the corresponding enumeration:

- **RadarLabelVisibility.First** - The annotation labels appears only on the first radial axis(default).
- **RadarLabelVisibility.All** - The annotation labels appears only on each radial axis(default).
- **RadarLabelVisibility.None** - No labels. The property has effect only on radar chart.

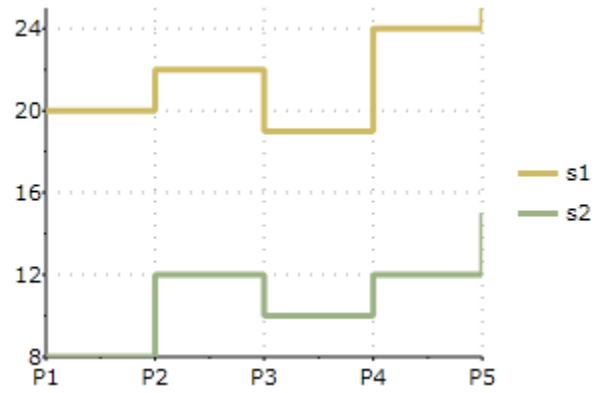
# Step Chart

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.

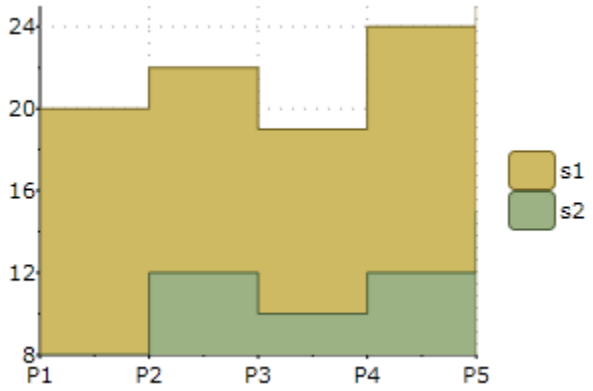
Similar to Line and XY plots, the appearance of the step chart can be customized by using the Connection and Symbol properties for each series by changing colors, symbol size, and line thickness. Symbols can be removed entirely to emphasize the relationship between points or included to indicate actual 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.

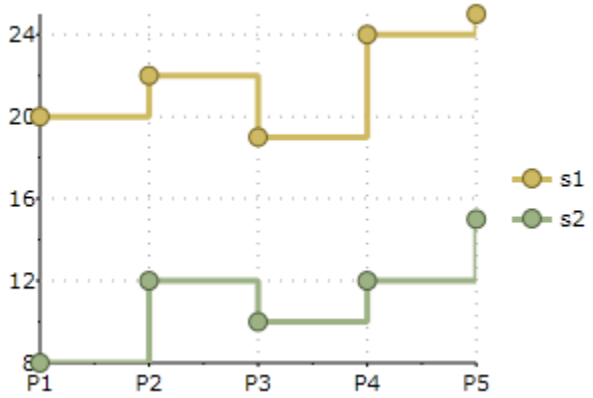
The following image represents the Step chart when you set ChartType property to **Step**.




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



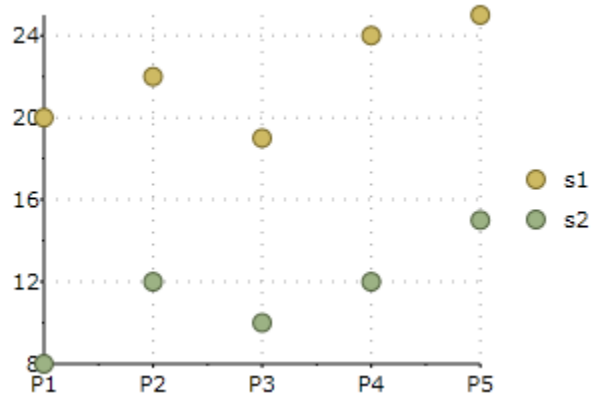

The following image represents the filled Step chart when you set ChartType property to **StepSymbols**.




# XYPlot Chart

The XYPlot is also known as a Scatter plot chart. For more information on the XYPlot chart see XY Charts (page 38).

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

# 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>
```

# 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. |
| AxisX, AxisY, **ChartView.AxisZ** | Each of these properties returns Axis objects that allow you to customize the appearance of the chart axes. |
| **ChartView.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. |

## Axes

The axes are represented by sub-properties of the View property: AxisX and AxisY. 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 76). |
| 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 80). |
| 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 76). |
| 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 83). |
| 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 79). |
| 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.

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.

| 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. |

## Dependent Axis

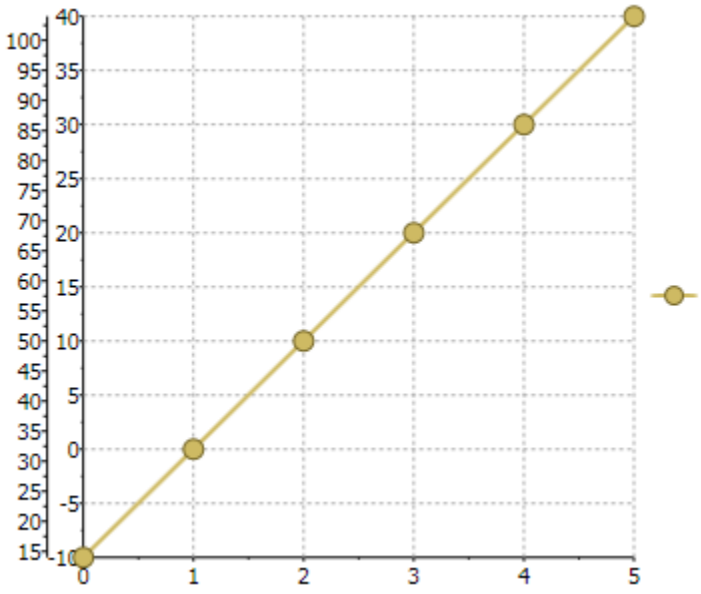
The IsDependent allows to link the auxiliary axis with one from the main axes(AxisX or AxisY, depending on AxisType). The dependent axis always has the same minimum and maximum as the main axis.

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 Fahrenheits 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 its 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.

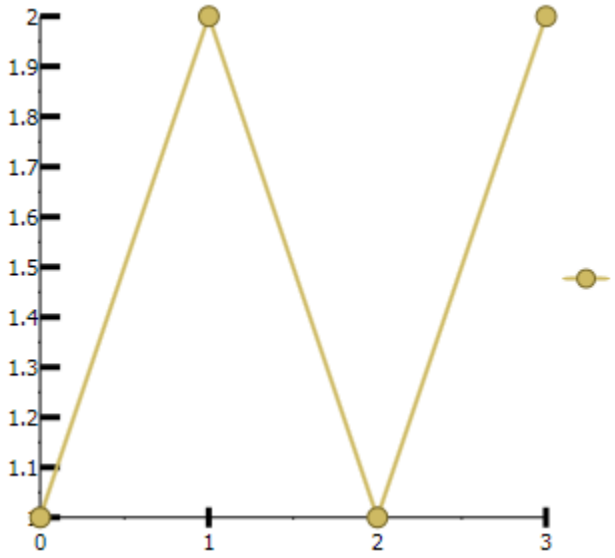
The MajorUnit and MinorUnit properties set the state of the Axis' tick marks. To eliminate clutter in a chart, you can display fewer labels or tick marks on the category (x) axis by specifying the intervals at which you want categories to be labeled, or by specifying the number of categories that you 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 not 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 descrease 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:

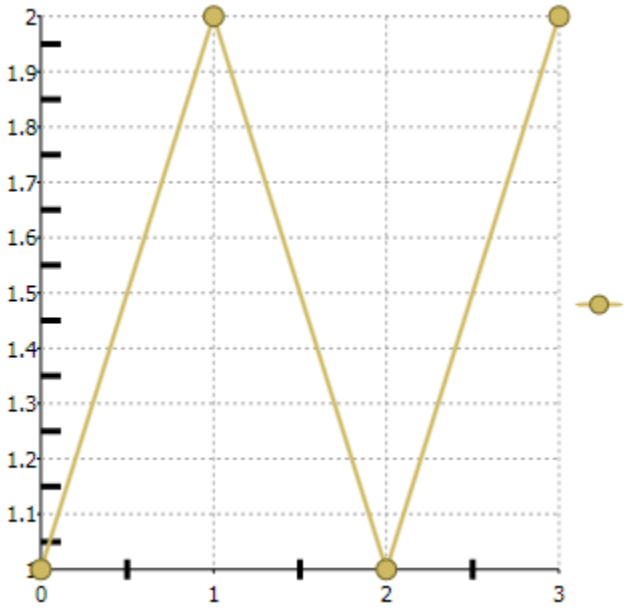



### *Minor Tick Overlap*

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 not 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/minor grid lines using the Axis.MajorGridFill/Axis.MinorGridFill 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.

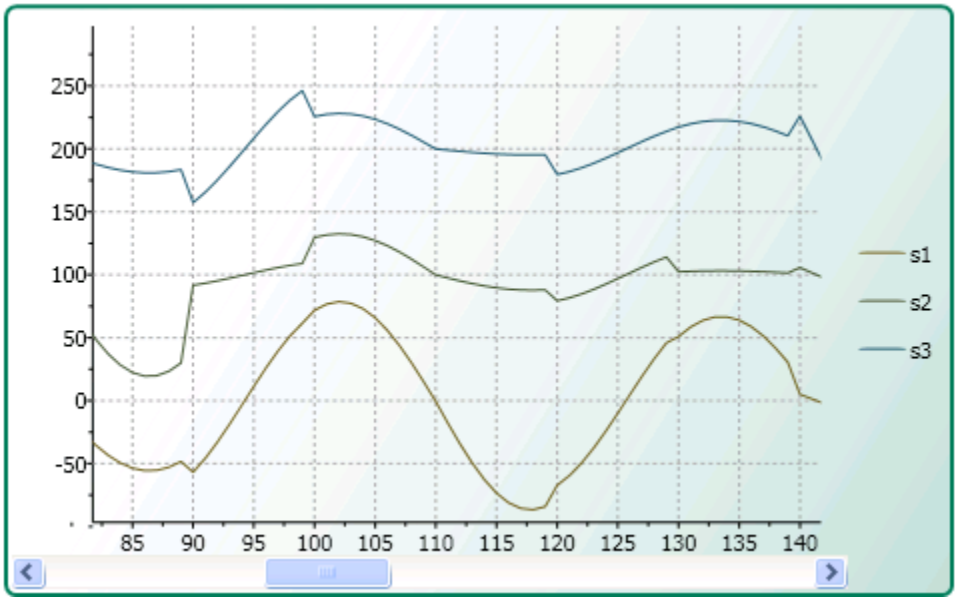
**Axis Min and Max**

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 a scrollbar to the axes. 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:

```
<ScrollBar Name="sb" DockPanel.Dock="Bottom" Minimum="0" Maximum="1"
                       SmallChange="0.1" LargeChange="0.2"
ViewportSize="0.2"
                       Orientation="Horizontal"
                       Value="{Binding
ElementName=c1Chart1,Path=View.AxisX.Value}" />
```

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

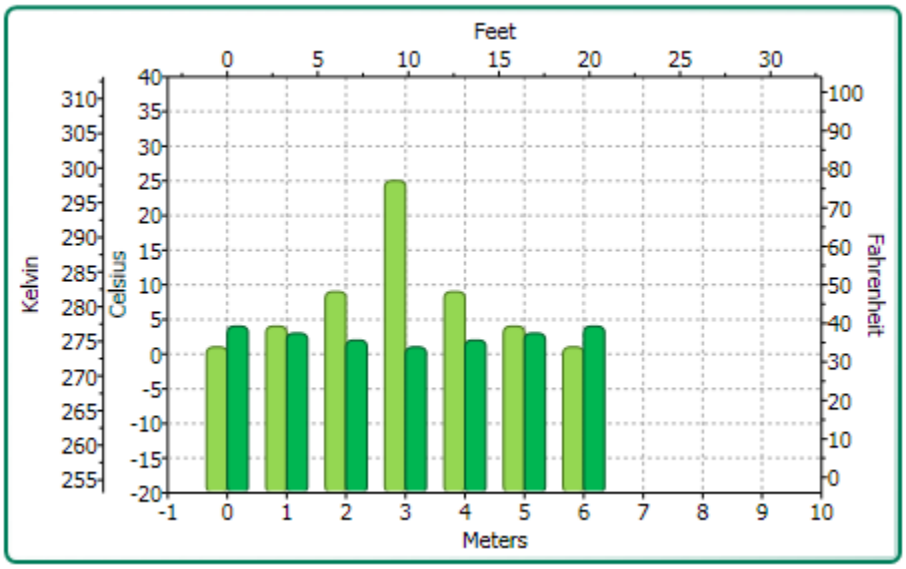
the Y-axis. Setting the Reversed property of the Axis, however will juxtapose these Maximum and Minimum values.

## 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 Axis.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>
```

## 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. |
| 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 Axis.AnnoFormat property see Standard Numeric Format Strings. For more information on the

**DateTime Format Strings**

The DateTime format strings are divided into two categories:

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

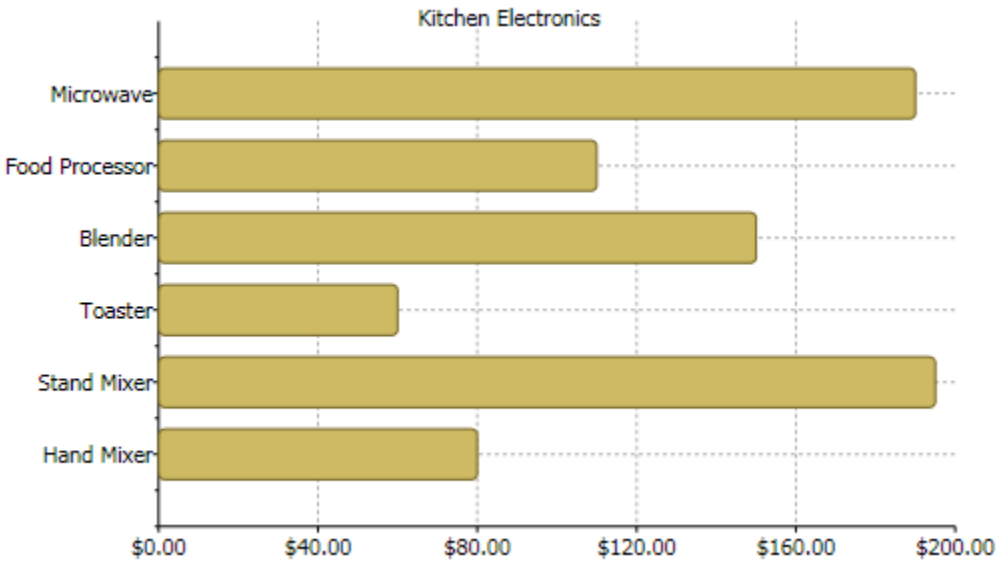
**Numeric Format Strings**

- 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 simply specifiy 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**

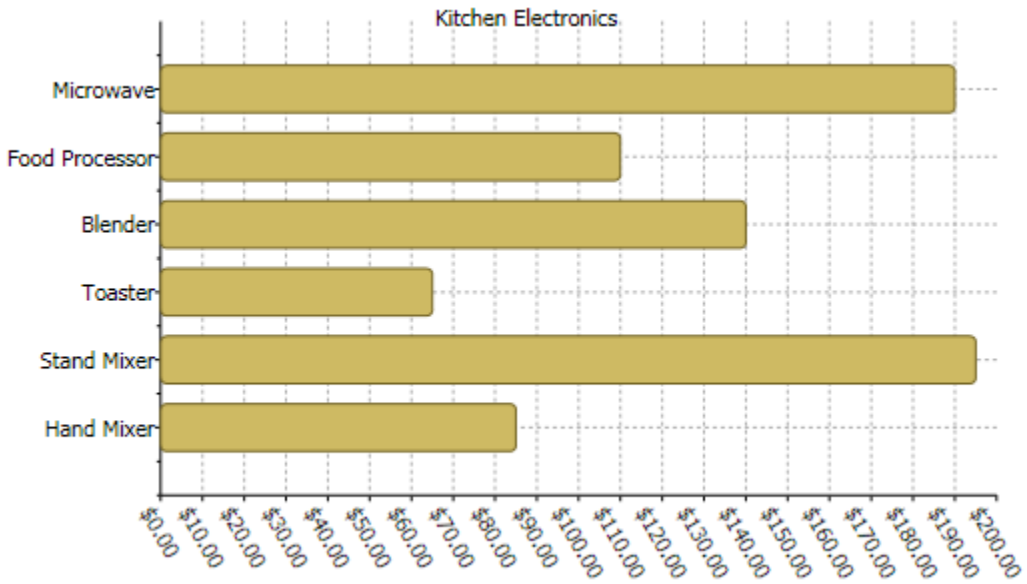
```
<c1chart:C1Chart.View>
                <c1chart:ChartView>
                    <c1chart:ChartView.AxisX>
                        <c1chart:Axis Min="0" AnnoFormat="c"
AutoMin="false" AutoMax="false" Max="200" />
                    </c1chart:ChartView.AxisX>
                </c1chart:ChartView>
 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

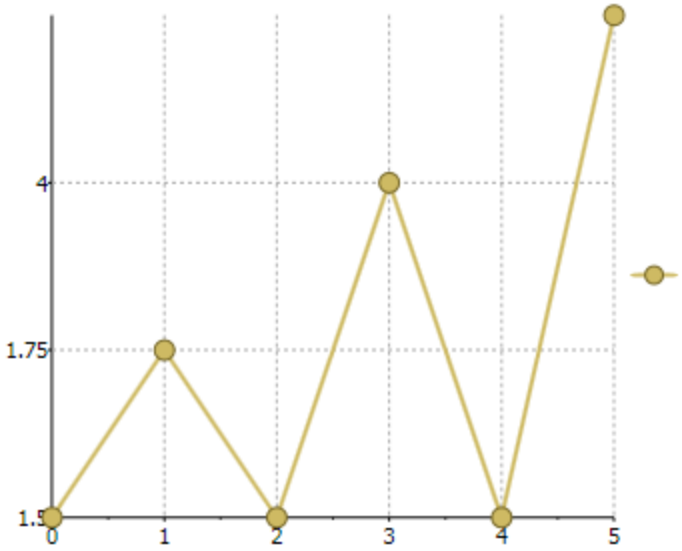
## Custom Axis Annotation

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 };
```

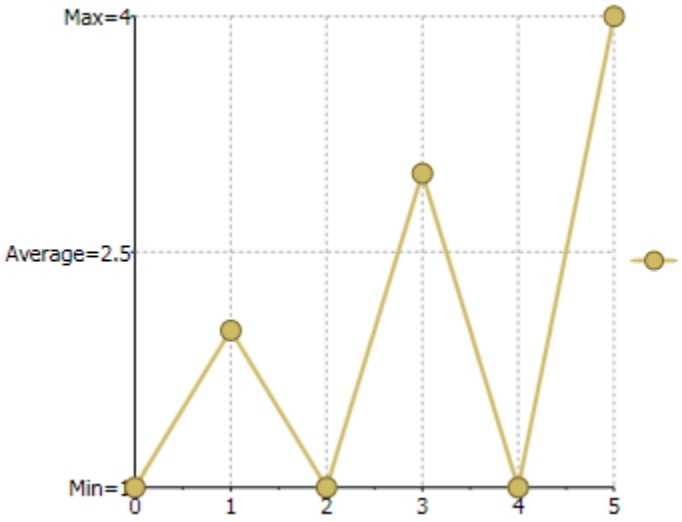
Here is what the chart appears like after adding the preceding code:




- When the ItemsSource property is a collection of KeyValuePair<object, double> or KeyValuePair<object, DateTime> the chart creates axis label 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)};
```

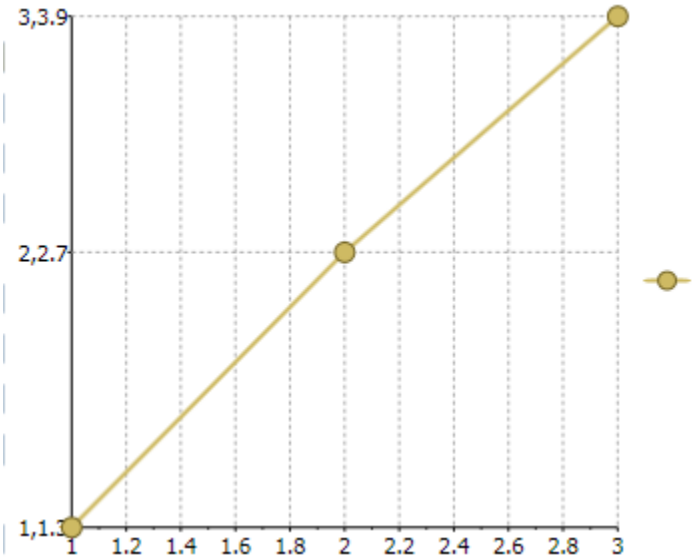
Here is what the chart appears like after adding the preceding code:



- 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:

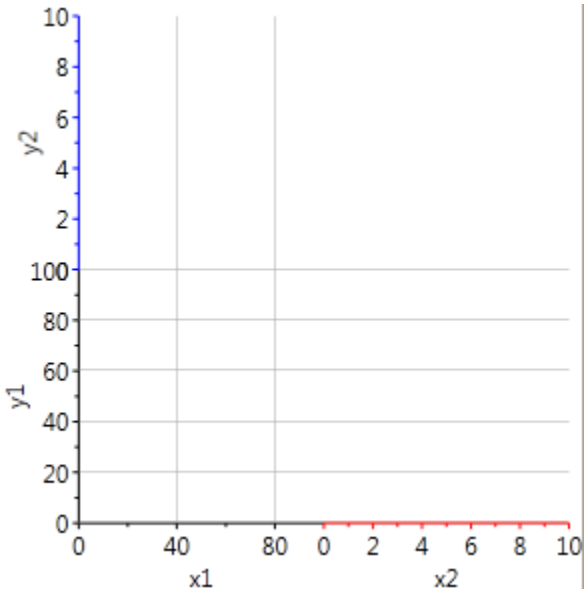



# Plot Area

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 example 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.
3. 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>
```

# 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 in XAML or programmatically through the Actions collection.

The following list reveals the supported chart actions:

- Scale action increases or decreases the scale of 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 1.0. 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 equal to 1.

- Zoom action allows the user to select rectangular area for viewing.

The scaling, translation and zooming are available only for chart with Cartesian axes.

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.

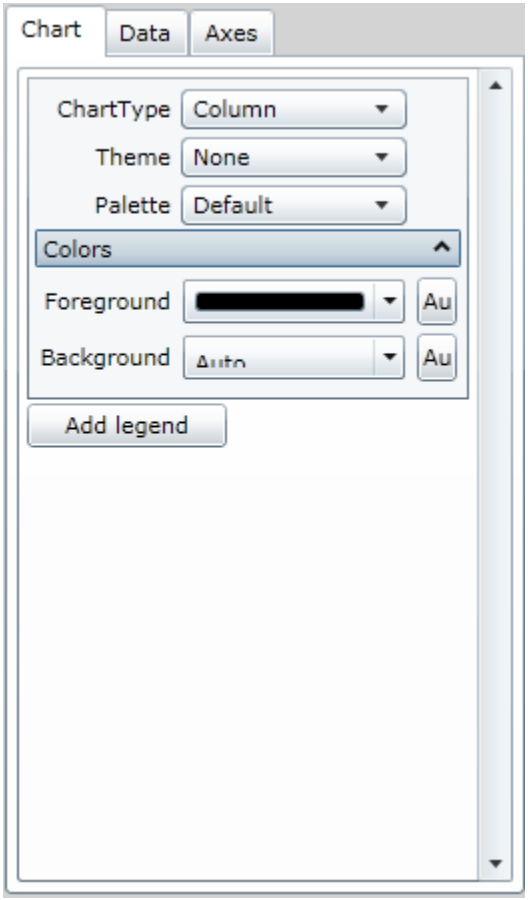
# Run-Time Chart Editor

This section explains how to get the chart editor to appear at run-time and it describes its graphical user interface.

The **Chart Editor** provides an easy and interactive way to create and modify a new or existing chart.

The **Chart Editor** can be accessed at run-time when you reference the **C1.Silverlight.Chart.Editor.dll** to your silverlight project and add the following line of XAML:

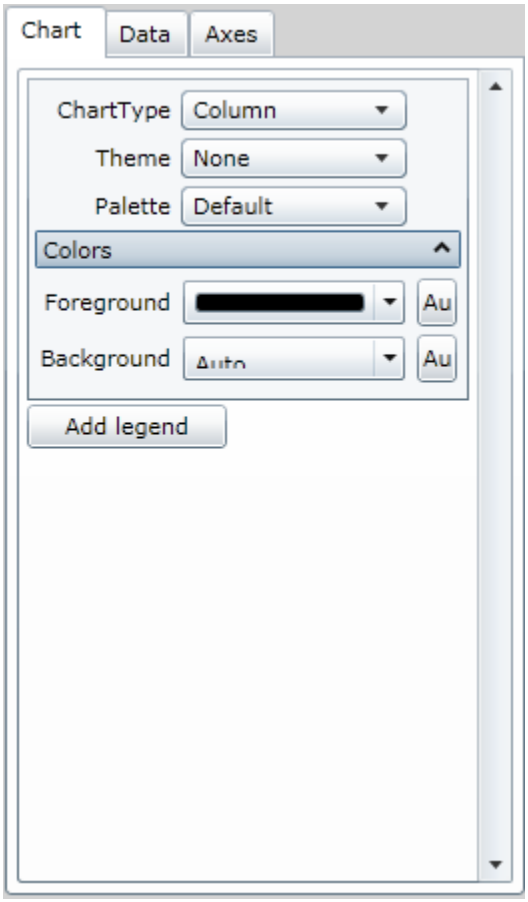
```
<c1charted:ChartEditor x:Name="ced" DataContext="{Binding
ElementName=chart}" />
```



The following topics provide a visual detail of the **Chart Editor** interface and explain the functionality of each element in the **Chart Editor**.

# Chart Tab

In the **Chart** tab you can choose from one of many chart types and then you customize the chart's appearance by choosing a theme, palette, foreground color, backround color, and legend.




# Data Tab

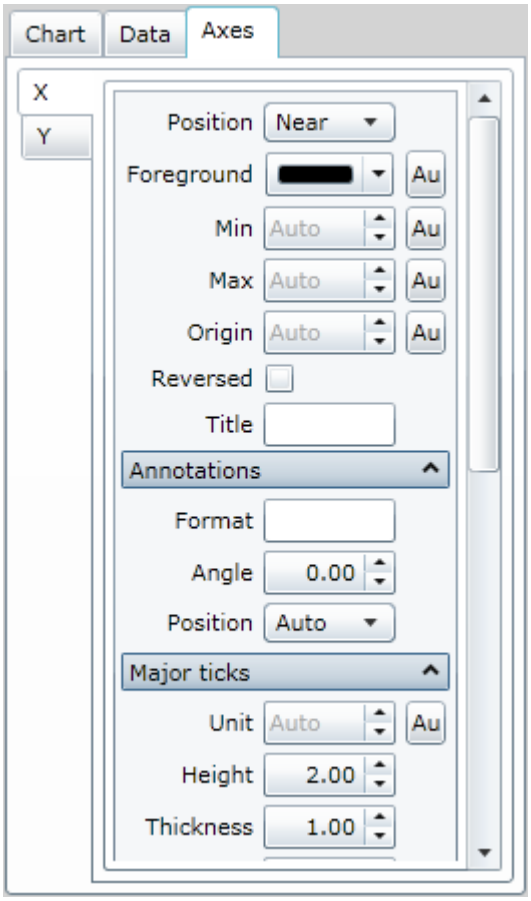
In the Data tab you can add and remove point labels and point tooltips to C1Chart. To add point labels and/or point tooltips click the associated **Add** button. To remove point labels and/or point tooltips click the associated **Remove** button.

Once a point label has been added you can change the value of its text, alignment, and offset.


## Axes Tab

You can select **Axes** tab to modify or create the format style, scale, and annotation of the axes.

The screen shot below represents the tab name for the selected Axis, in this example it is **Axis X**.




# Data-Binding

This section describes data binding.

The steps required to create data bound charts are identical to the ones we mentioned in the earlier topics

22. Choose the chart type (ChartType property).
23. Set up the axes (AxisX and AxisY properties).
24. Add one or more data series (Children collection).
25. Adjust the chart's appearance using the Theme and Palette properties.

    The only difference is in step 3. When you create data-bound charts, you need to set the ItemsSource property to the collection of items that contain the data you want to chart. Then, use the **dataSeries.ValueBinding** property to specify which property of the items contains the values to be plotted.

For example, here is the code we used before to generate the Sales Per Region chart (not data-bound):

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

// Show regions along label axis
_c1Chart.Data.ItemNames = (
  from r in data
  select r.Region).ToArray();

// 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.Data.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);
```

Here is the data-bound version of the code. The result is identical:

```
// Get the data
var data = GetSalesPerRegionData();
_c1Chart.Data.ItemsSource = data;

// Show regions along label axis
_c1Chart.Data.ItemNameBinding = new Binding("Region");

// Add data series
foreach (string series in "Revenue,Expense,Profit".Split(','))
{
  var ds = new DataSeries();
  ds.Label = series;
  ds.ValueBinding = new Binding(series);
  _c1Chart.Data.Children.Add(ds);
}
```

The data-bound version of the code is even more compact than the original. The three series are created in a loop, taking advantage of the fact that the names of the properties we want to chart are the same as the names we want to use for each data series.

You can assign any object that implements the **IEnumerable** interface to the **ItemsSource** property. This includes simple lists as shown above, LINQ queries, and **DataTable** objects provided by the **C1.Silverlight.Data** assembly.

# Data-Binding to C1.Silverlight.Data

**C1.Silverlight.Data** is an assembly that contains a subset of the data objects in ADO.NET built for the Silverlight platform (**DataSet**, **DataTable**, **DataView**, and so on). It allows you to re-use your ADO.NET data and logic in Silverlight applications, and integrates with LINQ and modern data-binding mechanisms as shown below.

Creating charts from **DataTable** objects is easy. The first step is obtaining the data from the server. Loading a **DataSet** from a database or file is discussed in detail in the **C1.Silverlight.Data** documentation. The code below shows one easy way to load a **DataSet** from an XML file:

```
public Page()
{
  InitializeComponent();

  // Other initialization
  // ...

  // Go read NorthWind product data
  WebClient wc = new WebClient();
  wc.OpenReadCompleted += wc_OpenReadCompleted;
```

```
    wc.OpenReadAsync(new Uri("products.xml", UriKind.Relative));
  }

  // Read NorthWind product data into data set
  DataSet _dataSet = null;
  void wc_OpenReadCompleted(object sender, OpenReadCompletedEventArgs
e)
  {
    _dataSet = new DataSet();
    _dataSet.ReadXml(e.Result);
  }
```

The code uses a **WebClient** object to obtain an XML stream from the server. It assumes the "products.xml" file is present on the server and contains the **NorthWind** products table. The "products.xml" file was obtained by calling the **DataSet.WriteXml** method.

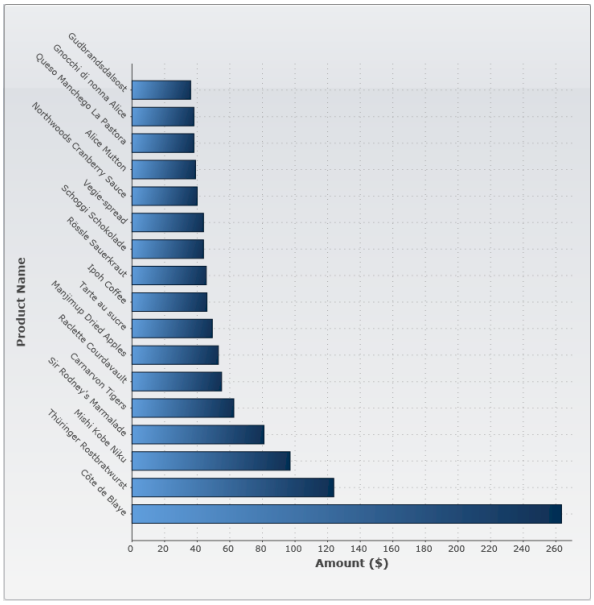
Once the data is available, creating the chart requires the same steps as before:

```
  void BuildNorthWindChart(ChartType chartType)
  {
    // Clear current chart
  _c1Chart.Reset(true);

    // Set chart type
    _c1Chart.ChartType = chartType;

    // Get axes
    Axis valueAxis = _c1Chart.View.AxisY;
    Axis labelAxis = _c1Chart.View.AxisX;
    if (chartType == ChartType.Bar)
    {
      valueAxis = _c1Chart.View.AxisX;
      labelAxis = _c1Chart.View.AxisY;
    }

    // Configure label axis
    labelAxis.Title = CreateTextBlock("Product Name", 14,
FontWeights.Bold);
    labelAxis.AutoMin = true;
    labelAxis.AnnoAngle = 45;

    // Configure value axis
    valueAxis.Title = CreateTextBlock("Amount ($)", 14,
FontWeights.Bold);
    valueAxis.AutoMin = false;
    valueAxis.Min = 0;

    // Get the data
    DataView dv = _dataSet.Tables["Products"].DefaultView;
    dv.RowFilter = "UnitPrice >= 35";
    dv.Sort = "UnitPrice DESC";
    _c1Chart.Data.ItemsSource = dv;

    // Show Product Name along x axis
    _c1Chart.Data.ItemNameBinding = new Binding("ProductName");

    // Add data series
    var ds = new DataSeries();
    ds.Label = "Unit Price";
```

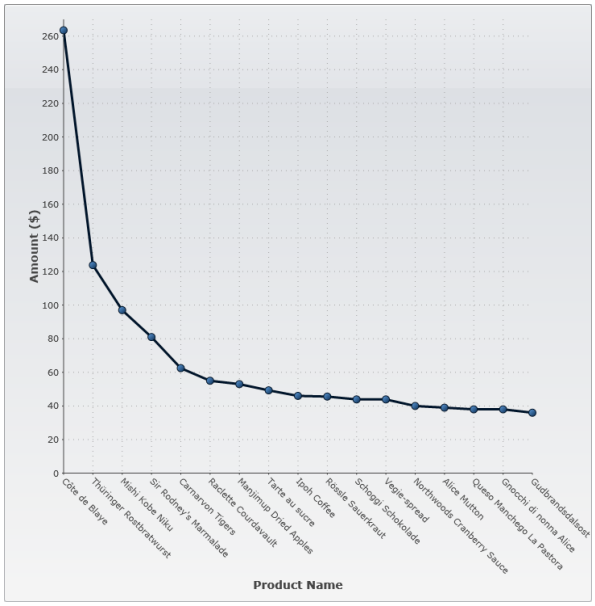
```
    ds.ValueBinding = new Binding("UnitPrice");
    _c1Chart.Data.Children.Add(ds);
}
```

The new code gets the data by retrieving the default view for the products table. It then applies a filter to show only products with prices over $35 per unit, and sorts the products by unit price.

This is the result:




BuildProductPriceChart(
  **ChartType.Bar**)




BuildProductPriceChart(
  **ChartType.LineSymbols**)

Alternatively, you could use LINQ to sort and filter the data in the **DataTable**:

```
    // Get the data
    //DataView dv = _dataSet.Tables["Products"].DefaultView;
    //dv.RowFilter = "UnitPrice >= 35";
    //dv.Sort = "UnitPrice DESC";
    //_c1Chart.Data.ItemsSource = dv;
    _c1Chart.Data.ItemsSource =
    (
      from dr
      in _dataSet.Tables["Products"].Rows
      where (decimal)dr["UnitPrice"] >= 35
      orderby dr["UnitPrice"] descending
      select dr.GetRowView()
    ).ToList();
```

The result is exactly the same as before. The difference between the two approaches is that the **DataView** object maintains a live connection to the underlying data. If the unit price of a product changed, the **DataView** would be automatically updated to reflect the new value and possibly filter the changed product in or out of view. The LINQ query, on the other hand, is converted into a static list and does not change.

# Data Labels and Tooltips

Data labels (also called data marker labels) are labels associated with data points. 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.

Going back to our previous example, let us add a simple label to each of the three data series. The first step would be to define the template as a resource in the **MainPage.xaml** file:

```
<UserControl x:Class="ChartIntro.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:c1chart="clr-
namespace:C1.Silverlight.Chart;assembly=C1.Silverlight.Chart">

  <!-- DataTemplate for chart labels -->
  <UserControl.Resources>
    <DataTemplate x:Key="chartLabel">
      <Border
        c1chart:PlotElement.LabelAlignment="MiddleCenter"
        Opacity="0.5" Background="White" CornerRadius="20"
        BorderBrush="Black" BorderThickness="2" Padding="4">
        <TextBlock
          Text="{Binding Value}"
          Foreground="Black" FontSize="16"/>
      </Border>
    </DataTemplate>
  </UserControl.Resources>

  <!-- Main grid -->
  <Grid x:Name="LayoutRoot" Background="White" Margin="10" >
    <!—no changes here -->
  </Grid>
</UserControl>
```

This resource defines how to obtain a data label for each point. Note that the template contains a **TextBlock** element with the **Text** property set to "{Binding Value}". This causes the text to be bound to the value of the associated data point. Also note that the **LabelAlignment** property is used to determine where the label should be positioned relative to the data point on the chart.

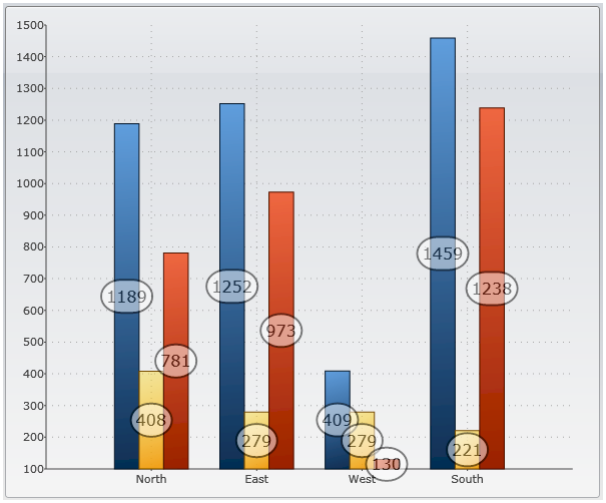
In order to use the new template, we need to modify the code in the **MainPage.xaml.cs** file as follows:

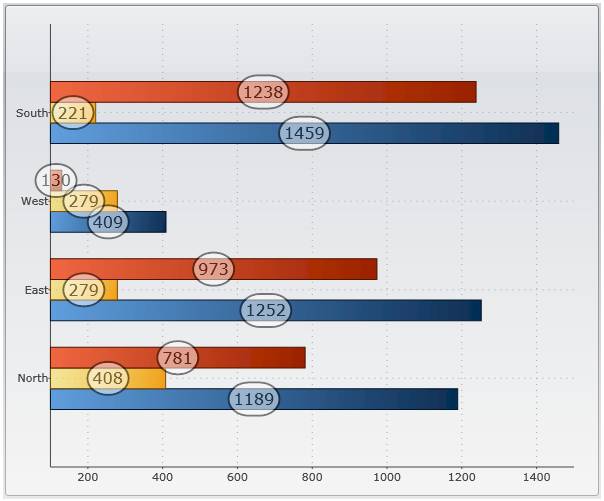
```
  // Get the data
  var data = GetSalesPerRegionData();
  _c1Chart.Data.ItemsSource = data;

  // Show regions along label axis
  _c1Chart.Data.ItemNameBinding = new Binding("Region");

  // Add data series
  foreach (string series in "Revenue,Expense,Profit".Split(','))
  {
    var ds = new DataSeries();
    ds.Label = series;
    ds.ValueBinding = new Binding(series);
    ds.PointLabelTemplate = Resources["chartLabel"] as DataTemplate;
    _c1Chart.Data.Children.Add(ds);
```

```
    }
```

The only change is one extra line of code that sets the PointLabelTemplate property to the resource defined in XAML.

Here is what the chart looks like after adding the labels:




You are not limited to showing a single value in each data label. **C1Chart** provides a DataPointConverter class that you can use to create more sophisticated bindings for your label templates. The converter is declared and used as a resource, along with the template.

For example, here is a revised version of the resource in the XAML file:

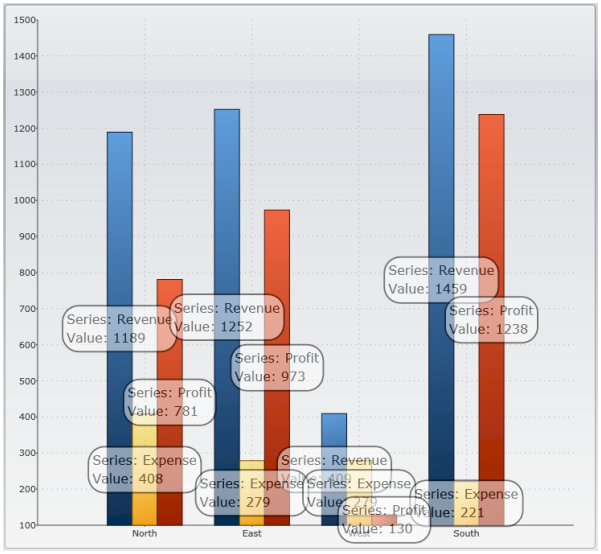
```
<UserControl x:Class="ChartIntro.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:c1chart="clr-
namespace:C1.Silverlight.Chart;assembly=C1.Silverlight.Chart">

  <!-- DataTemplate for chart labels -->
  <UserControl.Resources>
    <c1chart:DataPointConverter x:Key="dataPointConverter" />
    <DataTemplate x:Key="chartLabel">
      <Border
        c1chart:PlotElement.LabelAlignment="MiddleCenter"
        Opacity="0.5" Background="White" CornerRadius="20"
        BorderBrush="Black" BorderThickness="2" Padding="4">
        <TextBlock
         Text="{Binding
         Converter={StaticResource dataPointConverter},

ConverterParameter='{#SeriesLabel}{#NewLine}{#Value:$#,##0.00}'}"
          Foreground="Black" FontSize="16"/>
      </Border>
    </DataTemplate>
  </UserControl.Resources>

  <!-- Main grid -->
  <Grid x:Name="LayoutRoot" Background="White" Margin="10" >
    <!—no changes here -->
```

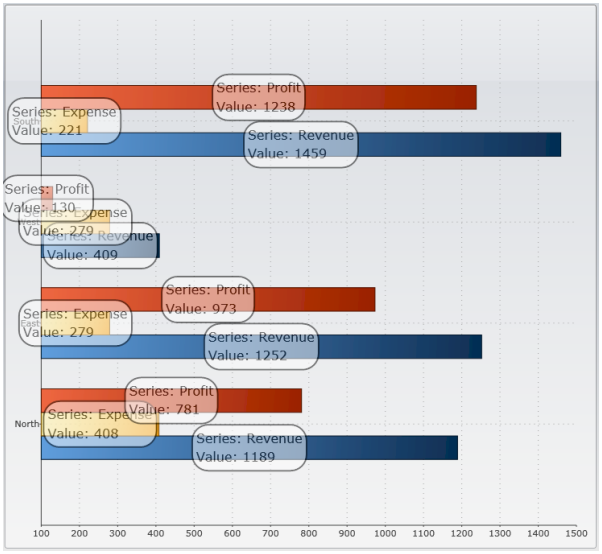
```
  </Grid>
</UserControl>
```

Notice the change in the binding for the **Text** property. This version specifies the converter and a converter parameter. The parameter is a string that contains keywords that will be replaced with data from the point that the label is bound to. The sample uses the **#SeriesLabel**, **#NewLine**, and **#Value** keywords. Other valid keywords are **#YValue**, **#PointIndex**, and **#SeriesIndex**.

The parameter also supports the usual .NET formatting syntax. The sample formats values as currencies with a currency symbol, thousand separators, and two decimals.

This is what the chart looks like after changing the template:




# Using MVVM

ComponentOne **Chart for Sil verlight** supports the MVVM (Model-View-ViewModel) design pattern. The entire chart can be declaratively written and bound to in XAML using native Silverlight 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 3: 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 Silverlight 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.

26. Add a **C1Chart** control to the page.
27. 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.

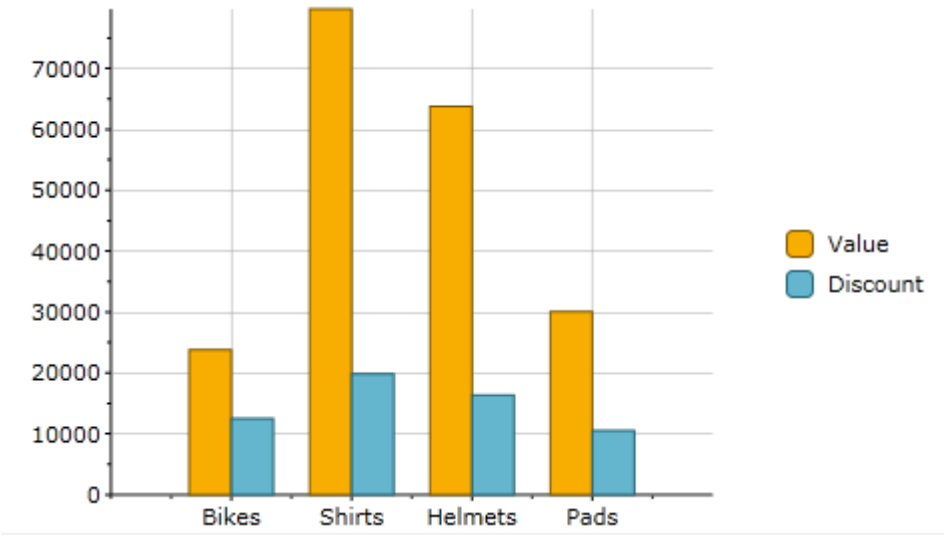
**Note:** If you are using **XYDataSeries**, then you should specify the XValueBinding for each series and you should not set the ItemNameBinding.

28. 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.

29. Run your application and observe that the **C1Chart** appears to be bound to the Sales data from the ViewModel.




# Advanced Topics

The following advanced topics detail using animations, zooming and panning, specialized charts, and XAML.

# Animation

Each DataSeries in a chart is composed of PlotElement objects that represent each individual symbol, connector, area, pie slice, and so on 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.

## OnLoad Animations

For example, the code below creates a 'fade-in' animation that causes each point in the data series to slowly appear, building the chart gradually:

```
void AnimateChart()
{
  // Build chart as usual
  SalesPerRegionAreaStacked_Click(this, null);

  // Make all series transparent and attach event handler
  // to make them visible gradually
  foreach (DataSeries ds in _c1Chart.Data.Children)
  {
    ds.Opacity = 0;
    ds.Loaded += ds_Loaded;
  }
}
```

The code starts by generating a chart as usual, and then loops through the DataSeries setting their **Opacity** to zero. This way, the chart will appear blank when it loads.

The code also attaches a handler to the **Loaded** event. This is where the animations will be added to each PlotElement. Here is the implementation:

```
// Animate each PlotElement after it has been loaded
void ds_Loaded(object sender, EventArgs e)
{
  PlotElement plotElement = sender as PlotElement;
  if (plotElement != null)
  {
    // Create storyboard to animate PlotElement
    Storyboard sb = new Storyboard();
    Storyboard.SetTarget(sb, plotElement);

    // Add Opacity animation to storyboard
    DoubleAnimation da = new DoubleAnimation();
    da.SetValue(Storyboard.TargetPropertyProperty, new
PropertyPath("Opacity"));
    da.Duration = new Duration(TimeSpan.FromSeconds(2));
    da.To = 1;
    sb.Children.Add(da);

    // Offset BeginTime for each series and point within series
    double seconds = 2;
    var dp = plotElement.DataPoint;
    if (dp != null && dp.PointIndex > -1)
    {
      seconds = dp.SeriesIndex + dp.PointIndex * 0.1;
    }
    da.BeginTime = TimeSpan.FromSeconds(seconds);
```

```
    // Start storyboard
    sb.Begin();
  }
}
```

This event handler gets called once for each PlotElement that is generated. The code creates a **Storyboard** object for each PlotElement and uses it to gradually change the opacity of the element from zero to one (completely transparent to completely solid).

Notice how the code uses the DataPoint property to determine which series and which data point the plot element belongs to, and then sets the **BeginTime** property of the animation to cause each plot element to become visible at different times. This way, the points appear one at a time, instead of all at once.

Notice also that the code tests the PointIndex property to make sure it is greater than -1. This is because some plot elements do not correspond to individual points, but rather to the whole series. This is the case for **Area** elements for example.

This code can be used for all chart types. You can use it to slowly show plot symbols, lines, pie slices, and so on.

### OnMouseOver Animations

You can also create animations that execute when the user moves the mouse over elements. To do this, use the Loaded event as before, but this time attach event handlers to each **PlotElement**'s mouse event instead of creating the animations directly.

For example:

```
void AnimatePoints()
{
  // Build chart as usual
  SalesPerMonthLineAndSymbol_Click(this, null);

  // Handle event when plot elements are created
  foreach (DataSeries ds in _c1Chart.Data.Children)
  {
    ds.Loaded += ds_Loaded;
  }
}

// Attach mouse event handlers to each plot element
// as they are created
void ds_Loaded(object sender, EventArgs e)
{
  PlotElement pe = sender as PlotElement;
  if (pe != null && pe.DataPoint.PointIndex > -1)
  {
    pe.MouseEnter += pe_MouseEnter;
    pe.MouseLeave += pe_MouseLeave;
  }
}

// Execute animations when the mouse enters or leaves
// each plot element
void pe_MouseEnter(object sender, MouseEventArgs e)
{
  AnimateDataPoint(sender as PlotElement, 3, 0.2);
}
void pe_MouseLeave(object sender, MouseEventArgs e)
{
```

```
  AnimateDataPoint(sender as PlotElement, 1, 1);
}
```

This code attaches event handlers that get called when the mouse enters or leaves each plot element. Both handlers call the **AnimateDataPoint** method, which increases the scale quickly when the mouse is over the element and restores it slowly when the mouse leaves the element.

Here is the implemenation of the **AnimateDataPoint** method:

```
void AnimateDataPoint(PlotElement plotElement, double scale, double duration)
{
  // Get/create scale transform for the PlotElement
  var st = plotElement.RenderTransform as ScaleTransform;
  if (st == null)
  {
    st = new ScaleTransform();
    plotElement.RenderTransform = st;
    plotElement.RenderTransformOrigin = new Point(0.5, 0.5);
  }

  // Create Storyboard and attach it to transform
  var sb = new Storyboard();
  Storyboard.SetTarget(sb, st);

  // Animate X and Y scales
  foreach (string prop in new string[] { "ScaleX", "ScaleY" })
  {
    var da = new DoubleAnimation();
    da.To = scale;
    da.Duration = new Duration(TimeSpan.FromSeconds(duration));
    da.SetValue(Storyboard.TargetPropertyProperty, new PropertyPath(prop));
    sb.Children.Add(da);
  }

  // Start animation
  sb.Begin();
}
```

# Zooming and Panning

The **C1Chart** control has features that make it easy to add zooming and panning to charts. These features enable users to select specific ranges of data to see details that are hidden in charts that show all the data at once.

## Zooming and Panning with Two Charts

Good examples of zooming and panning charts can be found in the "Google Financial" site and in the "Stock Portfolio" sample included with the ComponentOne Studio for Silverlight. Both applications show two charts. The bottom chart displays all the data available, and includes a range selector that allows users to pick the range they are interested in. The top chart displays data for the selected range only, allowing users to zero-in on interesting parts of the chart.

To implement this type of application using **C1Chart**, you would add two charts and a range slider to the page (or to a user control). For example:

```
<UserControl x:Class="ChartIntro.ZoomChart"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:c1="clr-namespace:C1.Silverlight;assembly=C1.Silverlight"
  xmlns:c1chart="clr-
namespace:C1.Silverlight.Chart;assembly=C1.Silverlight.Chart"
```

```
  Loaded="ZoomChart_Loaded" >

  <Grid x:Name="LayoutRoot" Background="White">
    <Grid.RowDefinitions>
      <RowDefinition Height="4*" />
      <RowDefinition Height="*" />
    </Grid.RowDefinitions>

    <!-- Main chart (shows selected range) -->
    <c1chart:C1Chart x:Name="_c1MainChart" />

    <!-- Zoom chart (shows entire data range) -->
    <c1chart:C1Chart x:Name="_c1ZoomChart" Grid.Row="1" />

    <!-- Range slider (selects a range from the zoom chart) -->
    <c1:C1RangeSlider x:Name="_slider" Grid.Row="1"
VerticalAlignment="Bottom"
         Minimum="0" Maximum="1" ValueChange="0.1"
         LowerValueChanged="_slider_ValueChanged"
         UpperValueChanged="_slider_ValueChanged" />
  </Grid>
</UserControl>
```

The XAML creates the controls and specifies two event handlers. The first event handler, **ZoomChart_Loaded**, is invoked when the page loads, and is responsible for initializing the charts:

```
// Draw main and zoom charts when the control loads
void ZoomChart_Loaded(object sender, RoutedEventArgs e)
{
  DrawChart(_c1MainChart);
  DrawChart(_c1ZoomChart);
}

// Draw a chart with some random data
void DrawChart(C1Chart chart)
{
  chart.Theme = ChartTheme.DuskGreen;
  chart.ChartType = ChartType.LineSymbols;

  var ds = new DataSeries();
  ds.ValuesSource = CreateData(100);
  chart.Data.Children.Add(ds);
}

// Create some random data for the chart
double[] CreateData(int cnt)
{
  var rnd = new Random(0);
  double[] data = new double[cnt];
  int last = 0;
  for (int i = 0; i < data.Length; i++)
  {
    int next = rnd.Next(0, 50);
    data[i] = last + next;
    last = next;
  }
  return data;
}
```

The code is similar to the one used in the first sections, when we introduced the basic **C1Chart** concepts. The routine that creates the chart is called twice, once to create the main chart above and once to create the zoom chart below it.

The next event handler is called when the user modifies the range using the range slider control. The event handler is responsible for updating the range displayed by the main chart:

```
// Update visible range in main chart when the slider changes
private void _slider_ValueChanged(object sender, EventArgs e)
{
  if ( c1MainChart != null)
  {
    Axis ax = _c1MainChart.View.AxisX;
    ax.Scale = _slider.UpperValue - _slider.LowerValue;
    ax.Value = _slider.LowerValue / (1 - ax.Scale);
  }
}
```

The event handler uses the Scale and **Axis.Value** properties to display the range selected by the user.

The Scale property determines *how much data* is shown on the chart. When **Scale** = 1, all data available is displayed; when **Scale** = 0.5, only half the data is displayed.

The Value property determines *what portion of the data* is displayed. When Value = 0, the initial part of the data is displayed; when Value = 1, the final part is displayed.

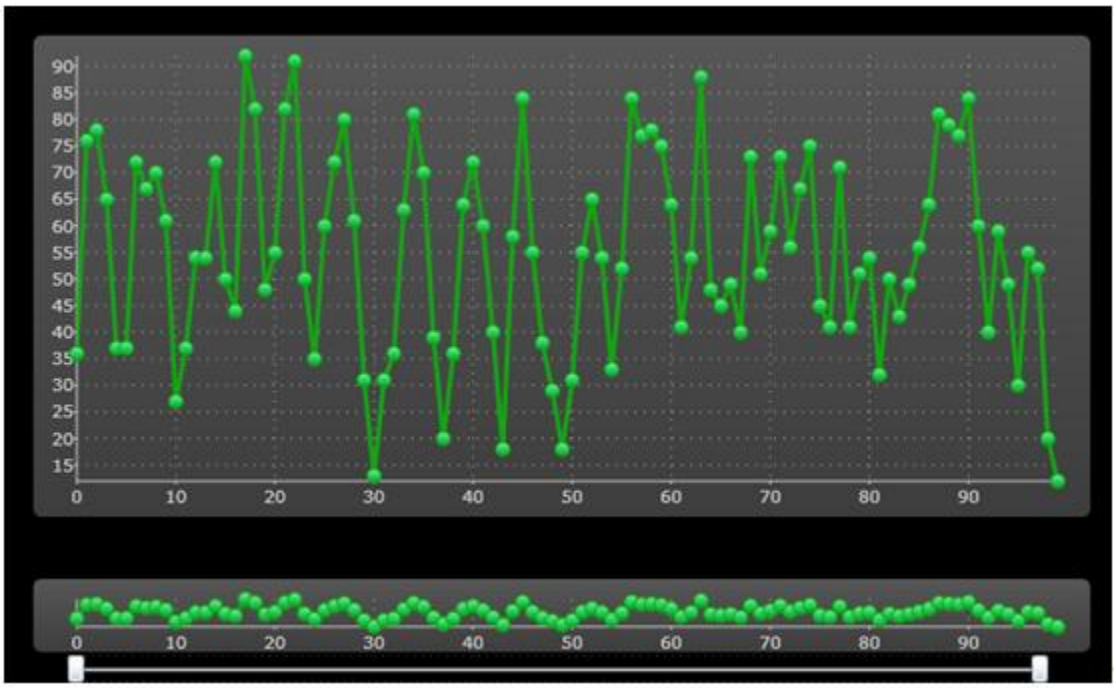
These properties are convenient but not strictly necessary. You could achieve the same results by setting the Max and Min properties on the axis. The advantage of using Scale and Value is that they work over a fixed range between zero and one, which makes the code simpler.

If you run the application now, you will see that it works correctly, but the range slider is aligned with the chart edges. We would like to align it with the edges of the plot area instead, so the relationship between the slider and the x axis is obvious to the user. Here is the code that aligns the range slider to the x axis:

```
// Set slider size to match the length of the x axis
protected override Size MeasureOverride(Size availableSize)
{
  Size sz = base.MeasureOverride(availableSize);
  _slider.Width = _c1MainChart.View.PlotRect.Width;
  return sz;
}
```

The code overrides the **MeasureOverride** method to set the width of the slider to match the width of the chart's plot rectangle (exposed by the PlotRect property).

If you run the project now, the result should look similar to the image below:




You can use the range slider below the bottom chart to select the range you are interested in. For example, if you dragged the slider's lower value to 55 and the upper value to 65, the upper chart would show the detail as in the image below:


You could improve this application by changing the template used by the range slider control. This is demonstrated in the "StockPortfolio" sample included with the ComponentOne Studio for Silverlight.

### Zooming and Panning with a Single Chart

You can also implement zooming and panning using a single chart. The easiest way to achieve this is to use the Actions property and the built-in mouse actions. **C1Chart** includes built-in actions for zooming, scaling, and panning the chart with the mouse.

The code below shows how you can add mouse actions to a **C1Chart** control:

```
// Show chart as usual
SalesPerRegionBar_Click(this, null);

// Add zoom action to mouse drag
_c1Chart.Actions.Add(new ZoomAction());

// Add scale action to control-mouse drag
_c1Chart.Actions.Add(new ScaleAction() { Modifiers = ModifierKeys.Control });

// Add pan action to shift-mouse drag
_c1Chart.Actions.Add(new TranslateAction() { Modifiers = ModifierKeys.Shift
});
```

Once the actions have been added to the **Actions** collection, you can use the mouse to perform the following actions:

- **Zoom**: Use the mouse to select a portion of the chart. When you release the mouse button, the chart will zoom in on the selected portion of the chart. This action allows you to zoom in but not to zoom out.
- **Scale**: Press the control button and move the mouse up or down to scale the chart interactively.
- **Pan**: Press the shift button and move the mouse to pan the display and see different portions of the chart without modifying the scale.

All built-in actions work by automatically setting the **Scale** property on each axis. You can limit or disable the actions by setting the **MinScale** property to one for either axis.

## Attaching Elements to Data Points

In previous sections, we discussed how you can customize the appearance of data series and individual data points. We also discussed how you can add data labels and tooltips to charts.

In some cases, however, you may need to add custom elements and position them relative to specific data points. For example, the Google Financials site displays charts with labels attached to specific data points. The labels relate the data points to significant news that affected the values on the chart.

This can be done easily with **C1Chart** using the PointFromData method. This method converts a specific point from chart coordinates to client coordinates, which you can use to position elements over the chart.

For example:

```
// Custom elements used to indicate maximum and minimum values
Ellipse _minMark, _maxMark;

// Create chart and initialize custom elements
void CustomElementChart()
{
  // Create chart as usual
  BuildSalesPerRegionChart(ChartType.LineSymbols);

  // Create min/max custom elements
  _minMark = new Ellipse()
  _minMark.Width = _minMark.Height = 45;
  _minMark.Stroke = new SolidColorBrush(Colors.Blue);
```

```
  _minMark.Fill = new SolidColorBrush(Color.FromArgb(64, 0, 0, 255));

  _maxMark = new Ellipse()
  _maxMark.Width = _maxMark.Height = 45;
  _maxMark.Stroke = new SolidColorBrush(Colors.Red);
  _maxMark.Fill = new SolidColorBrush(Color.FromArgb(64, 255, 0, 0));

  // Add custom elements to the chart
  _c1Chart.View.Children.Add(_minMark);
  _c1Chart.View.Children.Add(_maxMark);

  // Reposition custom elements when chart layout changes
  _c1Chart.LayoutUpdated += _c1Chart_LayoutUpdated;
}
```

The code starts by creating a chart as usual. Then it creates two **Ellipse** elements that will be used to indicate the minimum and maximum values on the chart. The custom elements are added to the View canvas. Finally, the code adds an event handler to the chart's **LayoutUpdated** event. The event handler is responsible for positioning the custom elements over data points on the chart whenever the chart layout changes (for example, when the chart is resized).

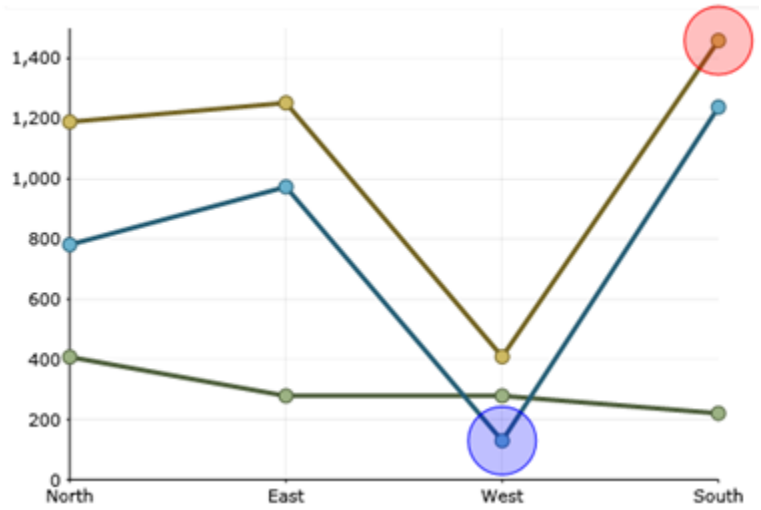
Here is the code for the event handler:

```
void _c1Chart_LayoutUpdated(object sender, EventArgs e)
{
  // Find minimum and maximum values
  int imax = 0, imin = 0;
  double ymin = double.MaxValue, ymax = double.MinValue;
  foreach (DataSeries ds in _c1Chart.Data.Children)
  {
    double[] values = ds.ValuesSource as double[];
    for (int i = 0; i < values.Length; i++)
    {
      if (values[i] > ymax)
      {
        ymax = values[i];
        imax = i;
      }
      if (values[i] < ymin)
      {
        ymin = values[i];
        imin = i;
      }
    }
  }

  // Position custom element over minimum
  Point ptMin = _c1Chart.View.PointFromData(new Point(imin, ymin));
  Canvas.SetLeft(_minMark, ptMin.X - _minMark.Width / 2);
  Canvas.SetTop(_minMark, ptMin.Y - _minMark.Height / 2);

  // Position custom element over maximum
  Point ptMax = _c1Chart.View.PointFromData(new Point(imax, ymax));
  Canvas.SetLeft(_maxMark, ptMax.X - _maxMark.Width / 2);
  Canvas.SetTop(_maxMark, ptMax.Y - _maxMark.Height / 2);
}
```

The event handler starts by scanning all the data series to find the maximum and minimum values on the chart and their index along the X axis.

Once these values have been found, the event handler calls the **PointFromData** method to convert the data coordinates into pixel coordinates within the chart's plot area. Finally, it calls the **SetLeft** and **SetTop** methods to position the custom elements so the center of the ellipses coincides with the point on the chart.

The result looks like the image below:




The red marker is positioned over the maximum value and the blue over the minimum. The position of the markers is updated automatically when the chart is resized.

# Using XAML

In this document, we have created several charts using C# code. But you can also create charts entirely in XAML and using Blend or Visual Studio. The advantage of doing this is you can create charts interactively and see the effect of each change immediately.

To show how this works, open a project that contains a reference to the **C1.Silverlight.Chart** assembly and add new Silverlight user control named "XamlChart" to the project. Then open the XAML file in Visual Studio or Blend and copy or type the following content into it:

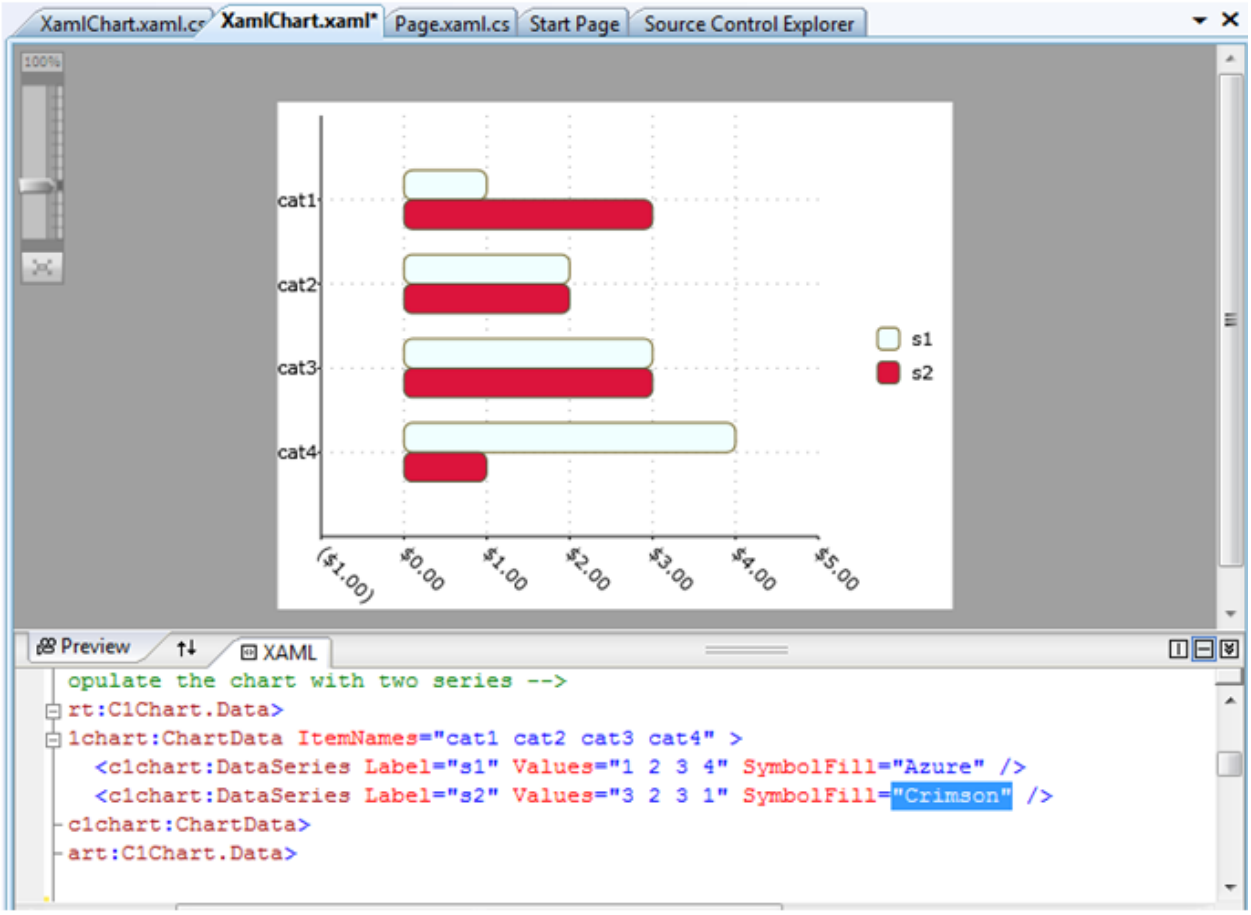
```
<UserControl x:Class="ChartIntro.XamlChart"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:c1chart="clr-
namespace:C1.Silverlight.Chart;assembly=C1.Silverlight.Chart"
  xmlns:c1="clr-namespace:C1.Silverlight;assembly=C1.Silverlight"
  Width="400" Height="300">
  <Grid x:Name="LayoutRoot" Background="White">

  <!-- Create chart, specify type -->
  <c1chart:C1Chart x:Name="chart0" ChartType="Bar" >

    <!-- Populate the chart with two series -->
    <c1chart:C1Chart.Data>
      <c1chart:ChartData ItemNames="cat1 cat2 cat3 cat4" >
        <c1chart:DataSeries Label="s1" Values="1 2 3 4" SymbolFill="Azure" />
```

```
        <c1chart:DataSeries Label="s2" Values="3 2 3 1" SymbolFill="Crimson"
/>
      </c1chart:ChartData>
    </c1chart:C1Chart.Data>

    <!-- Configure axes -->
    <c1chart:C1Chart.View>
      <c1chart:ChartView>
        <c1chart:ChartView.AxisX>
          <c1chart:Axis Min="-1" Max="5" AnnoFormat="c" AnnoAngle="45"
MajorUnit="1"/>
        </c1chart:ChartView.AxisX>
        <c1chart:ChartView.AxisY>
          <c1chart:Axis Reversed="True"/>
        </c1chart:ChartView.AxisY>
      </c1chart:ChartView>
    </c1chart:C1Chart.View>
  </c1chart:C1Chart>

  <!-- Add a legend -->
  <c1chart:C1ChartLegend />

  </Grid>
</UserControl>
```

If you edit this code in Visual Studio's split window, you will be able to see the effect of each change you make as you type it. This makes it easy to experiment with different settings to get the results you want.

Notice that you can edit most chart elements directly in XAML, including the data series and the axes.

The image below shows the interactive editing process within Visual Studio:

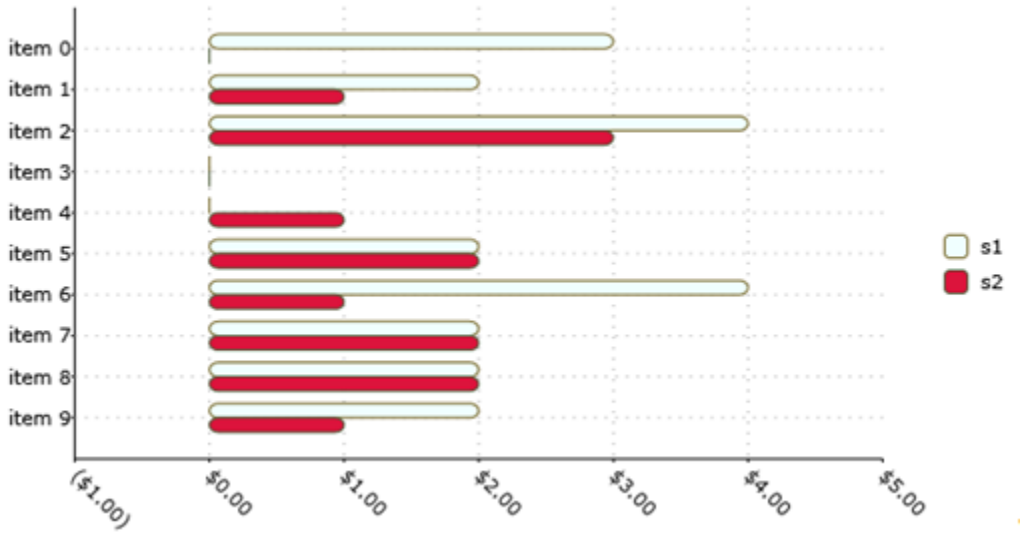



This is a convenient way to get the chart set up. At run time, you would use code to provide the actual chart data by setting the **ItemNames** and **ValuesSource** property on each data series as we did in earlier examples. For example:

```
public XamlChart()
{
  // Initialize control
  InitializeComponent();

  // Set item names and series values
  chart0.Data.ItemNames = GetItemNames(10);
  foreach (DataSeries ds in chart0.Data.Children)
  {
    ds.ValuesSource = GetSeriesData(10);
  }
}

string[] GetItemNames(int count)
{
  string[] names = new string[count];
  for (int i = 0; i < count; i++)
  {
    names[i] = string.Format("item {0}", i);
  }
  return names;
```

```
}
Random _rnd = new Random();
double[] GetSeriesData(int count)
{
  double[] values = new double[count];
  for (int i = 0; i < count; i++)
  {
    values[i] = _rnd.Next(0, 5);
  }
  return values;
}
```

The code replaces the dummy data we used at design time the actual data. The final result is shown below:




# 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.Silverlight.Chart.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:

30. 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.
31. 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.Silverlight.C1Chart.Extended.dll** in your project.

# Chart for Silverlight Appearance

The following topics detail how to customize the C1Chart control's appearance. Themes allow you to customize the appearance of the chart and take advantage of Silverlight's XAML-based styling. You can also use templates to format and layout the control and to customize the control's actions.

## XAML Elements

Several auxiliary XAML elements are installed with **ComponentOne Chart for Silverlight**. These elements include templates and themes and are located in the Chart for Silverlight installation directory, by default **C:\Program Files\ComponentOne\Studio for Silverlight\Help\C1.Silverlight.Chart**. 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 Silverlight** with their location within the C:\Program Files\ComponentOne\Studio for Silverlight\Help\C1.Silverlight.Chart folder noted:

| Element | Folder | Description |
|---|---|---|
| DuskBlue.xaml | ThemesSL | Specifies the templates for the Dusk Blue theme. |
| DuskGreen.xaml | ThemesSL | Specifies the templates for the Dusk Green theme. |
| generic.xaml | Themes | Specifies the templates for different styles and the initial style of the chart. |
| MediaPlayer.xaml | ThemesSL | Specifies the templates for the Media Player theme. |
| Office2003Blue.xaml | ThemesSL | Specifies the templates for the Office 2003 Blue theme. |
| Office2003Classic.xaml | ThemesSL | Specifies the templates for the Office 2003 Classic theme. |
| Office2003Olive.xaml | ThemesSL | Specifies the templates for the Office 2007 Olive theme. |
| Office2003Royale.xaml | ThemesSL | Specifies the templates for the Office 2007 Royal theme. |
| Office2003Silver.xaml | ThemesSL | Specifies the templates for the Office 2007 Silver theme. |
| Office2007Black.xaml | ThemesSL | Specifies the templates for the Office 2007 Black theme. |
| Office2007Blue.xaml | ThemesSL | Specifies the templates for the Office 2007 Blue theme. |
| Office2007Silver.xaml | ThemesSL | Specifies the templates for the Office 2007 Silver theme. |
| Vista.xaml | ThemesSL | Specifies the templates for the Vista theme. |

# 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 forground 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

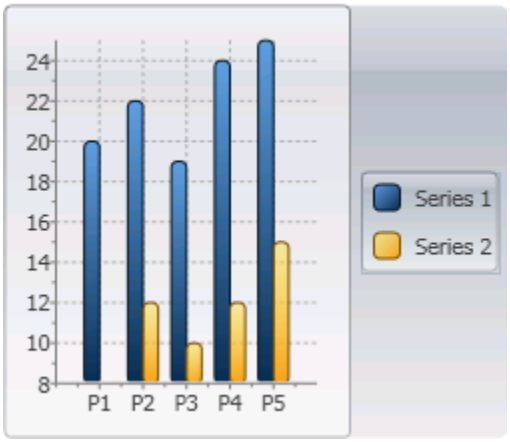
| Resource Key | Description |
|---|---|
| C1Chart_AxisMajorGridStroke_Brush | Represents the AxisMajorGridStroke's brush. |
| C1Chart_AxisMinorGridStroke_Brush | Represents the AxisMinorGridStroke's brush. |

# Chart Themes

**ComponentOne Chart for Silverlight** 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="Office2007Black”>
```

#### In Code

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

- Visual Basic

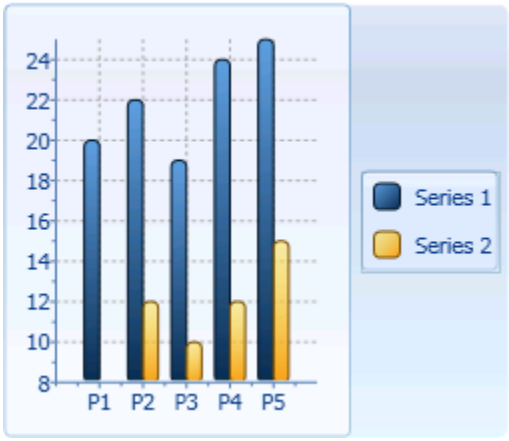
```
C1Chart1.Theme = ChartTheme.Office2007Black
```

- C#

```
c1Chart1.Theme = ChartTheme.Office2007Black;
```

### 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="Office2007Blue">
```

**In Code**

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

- Visual Basic

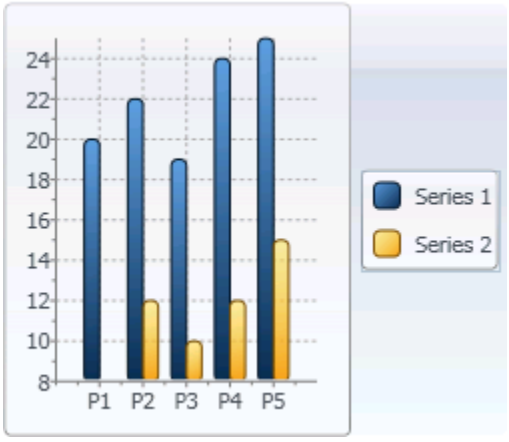
```
C1Chart1.Theme = ChartTheme.Office2007Blue
```

- C#

```
C1Chart1.Theme = ChartTheme.Office2007Blue;
```

**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="Office2007Silver">
```

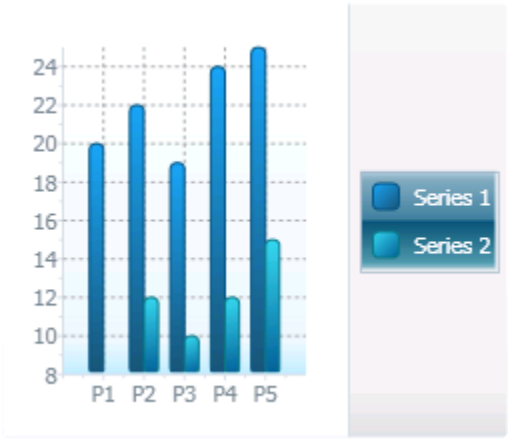
**In Code**

To specifically define the Office2007Silver theme in your chart, add the following code your project:

- Visual Basic
```
C1Chart1.Theme = ChartTheme.Office2007Silver
```
- C#
```
c1Chart1.Theme = ChartTheme.Office2007Silver;
```

**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="Vista">
```

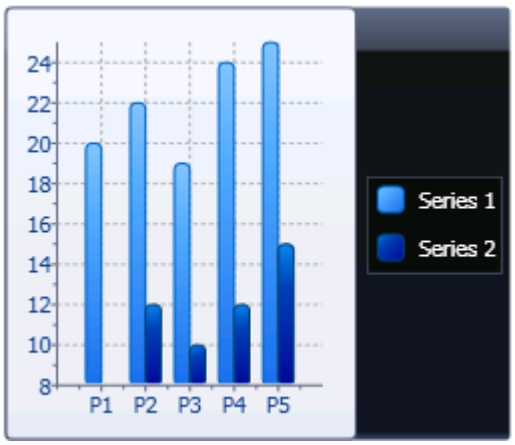
**In Code**

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

- Visual Basic
```
C1Chart1.Theme = ChartTheme.Vista
```
- C#
```
c1Chart1.Theme = ChartTheme.Visata;
```

**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="MediaPlayer">
```

**In Code**

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

- Visual Basic

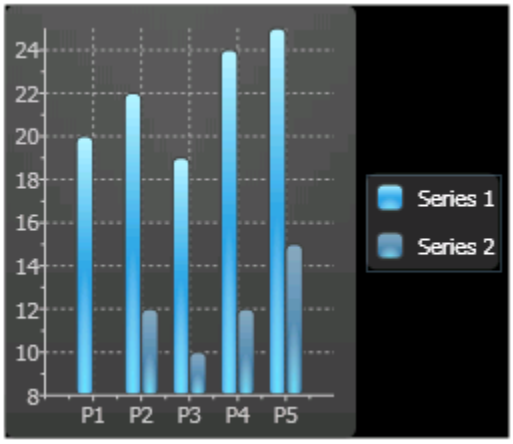
```
C1Chart1.Theme = ChartTheme.MediaPlayer
```

- C#

```
c1Chart1.Theme = ChartTheme.MediaPlayer;
```

**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="DuskBlue">
```

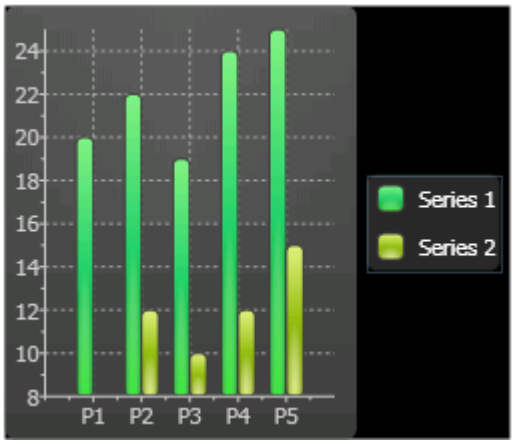
**In Code**

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

- Visual Basic
```
C1Chart1.Theme = ChartTheme.DuskBlue
```
- C#
```
c1Chart1.Theme = ChartTheme.DuskBlue;
```

**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="DuskGreen">
```

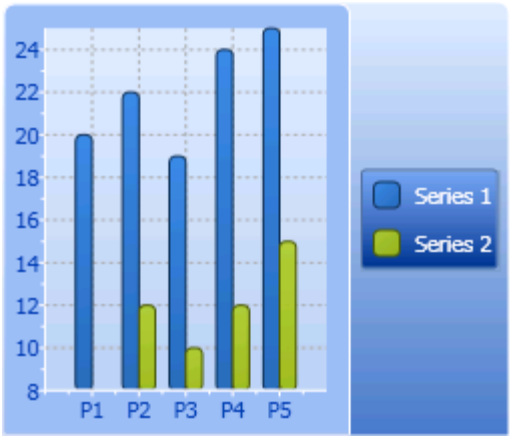
**In Code**

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

- Visual Basic
```
C1Chart1.Theme = ChartTheme.DuskGreen
```
- C#
```
c1Chart1.Theme = ChartTheme.DuskGreen;
```

**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="Office2003Blue">
```

**In Code**

To specifically define the Office2003Blue theme in your chart, add the following code your project:

- Visual Basic

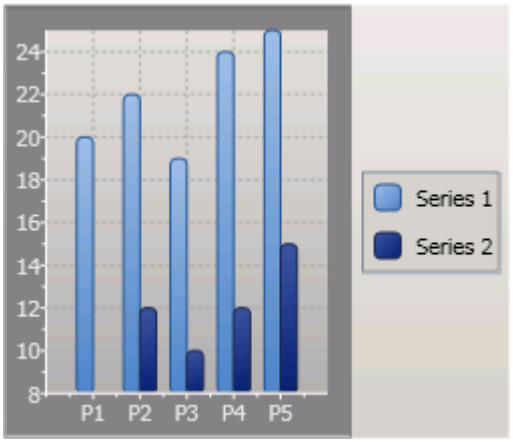
```
C1Chart1.Theme = ChartTheme.Office2003Blue
```

- C#

```
c1Chart1.Theme = ChartTheme.Office2003Blue;
```

**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="Office2003Classic">
```

**In Code**

To specifically define the Office2003Classic theme in your chart, add the following code your project:

- Visual Basic
```
C1Chart1.Theme = ChartTheme.Office2003Classic
```
- C#
```
c1Chart1.Theme = ChartTheme.Office2003Classic;
```

**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="Office2003Olive">
```

**In Code**

To specifically define the Office2003Olive theme in your chart, add the following code your project:

- Visual Basic
```
C1Chart1.Theme = ChartTheme.Office2003Olive
```
- C#
```
c1Chart1.Theme = ChartTheme.Office2003Olive;
```

**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="Office2003Royale">
```

**In Code**

To specifically define the Office2003Royale theme in your chart add the following code your project:

- Visual Basic

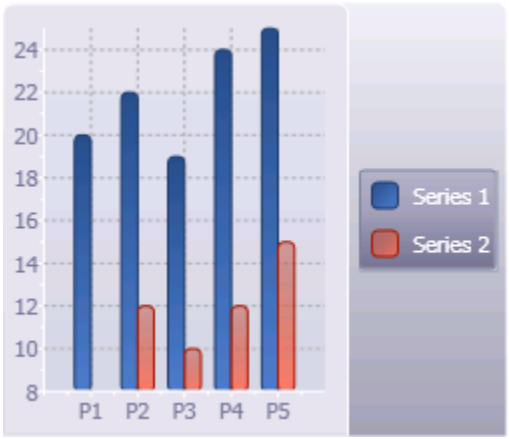
```
C1Chart1.Theme = ChartTheme.Office2003Royale)
```

- C#

```
c1Chart1.Theme = ChartTheme.Office2003Royale;
```

**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="Office2003Silver">
```

**In Code**

To specifically define the Office2003Silver theme in your chart, add the following code your project:

- Visual Basic

```
C1Chart1.Theme = ChartTheme.Office2003Silver)
```

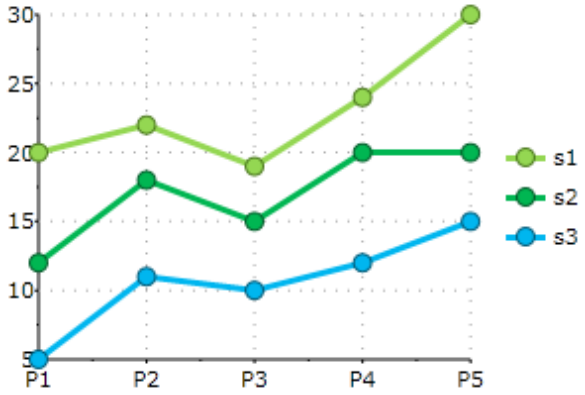
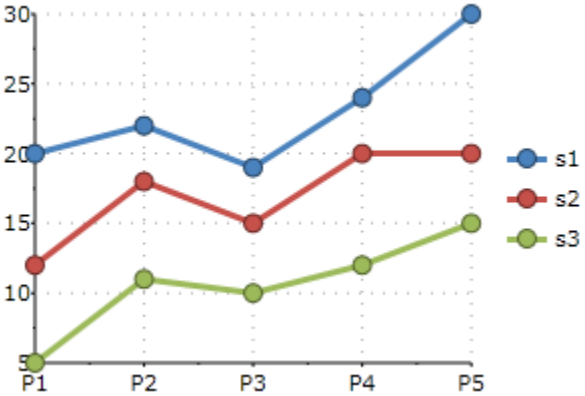
- C#

```
c1Chart1.Theme = ChartTheme.Office2003Silver;
```

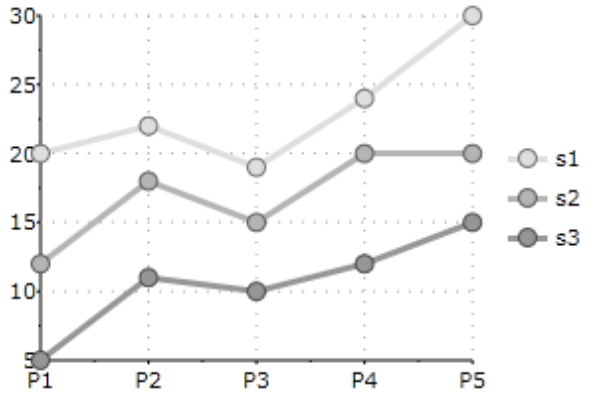
# Data Series Color Palette

The data series color scheme can be selected by using the Palette property. By default, C1Chart uses the Palette.**Custom** setting which specifies the standard color generation. The remaining options mimic the color themes of Microsoft Office.

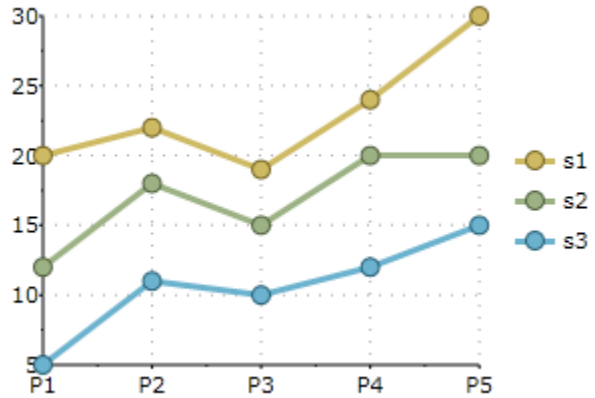
Available color schemes for the data series are listed below:

| **Color Generation Setting** | **Description or Preview** |
|---|---|
| Default | By default, C1Chart uses the Palette.Default. |
| Standard |  |
| Office |  |

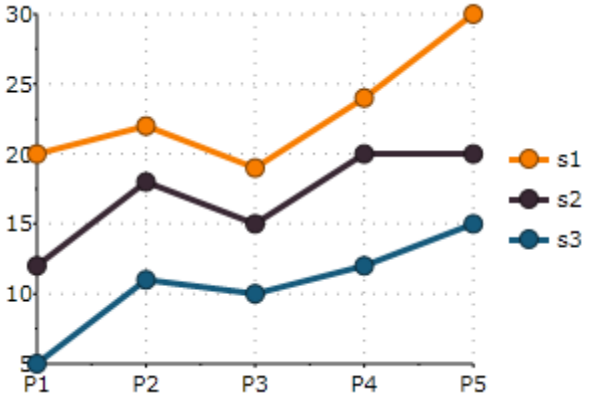
GrayScale



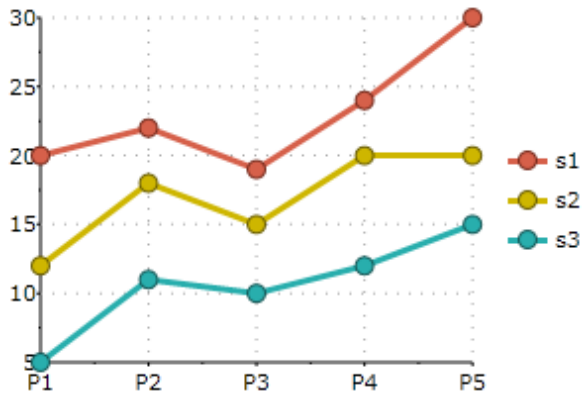

Apex



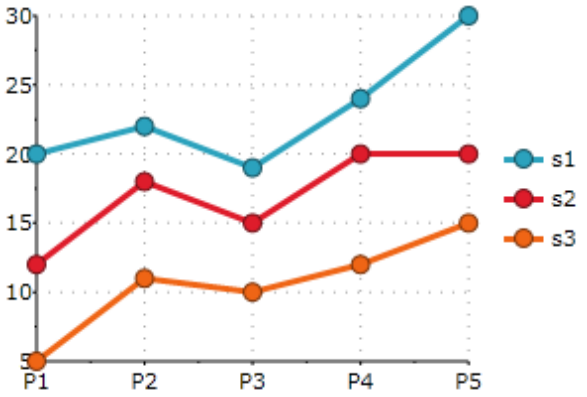

Aspect



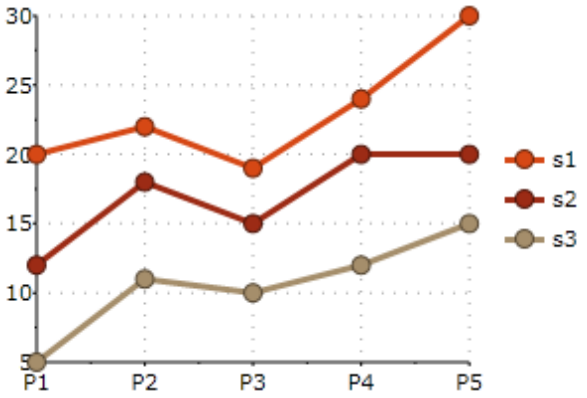
Civic



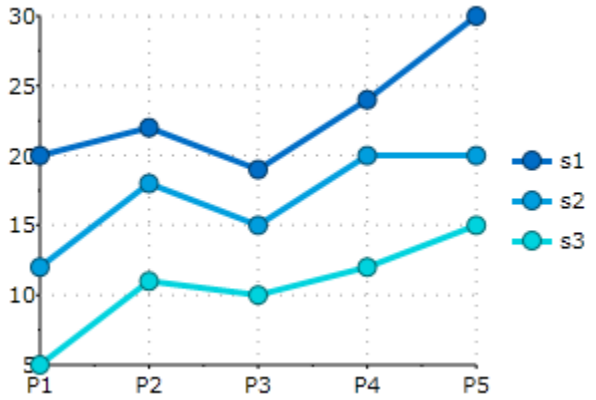

Concourse



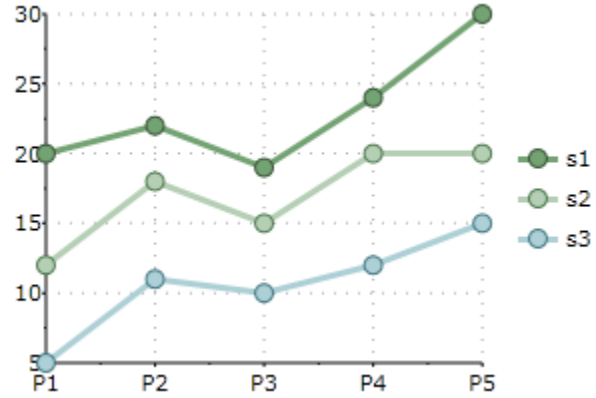

Equity



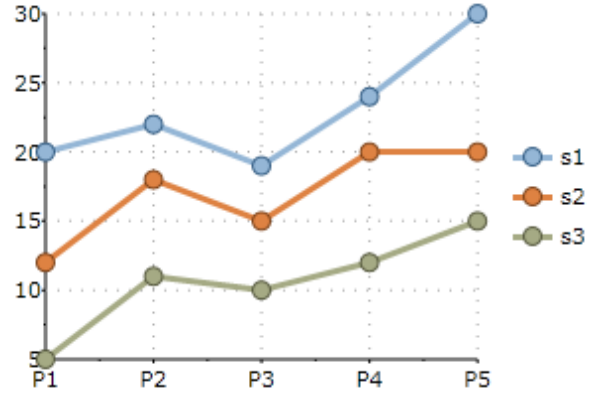
Flow



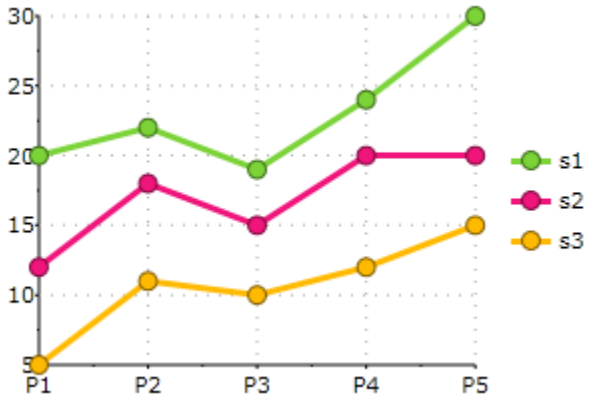

Foundry



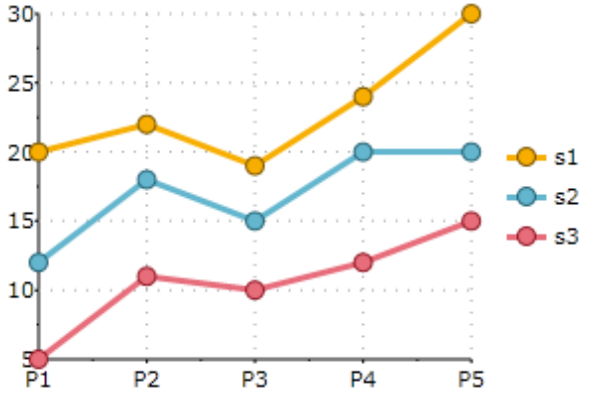

Median



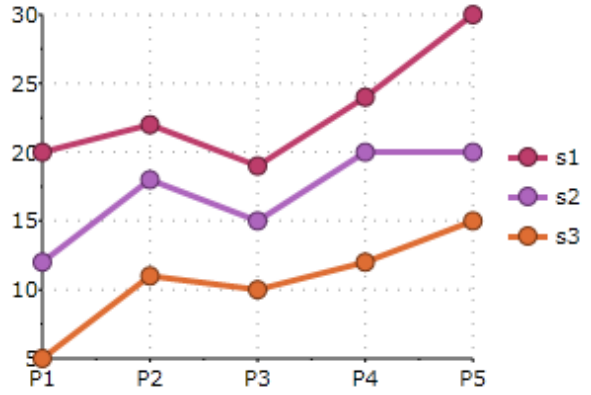
Metro



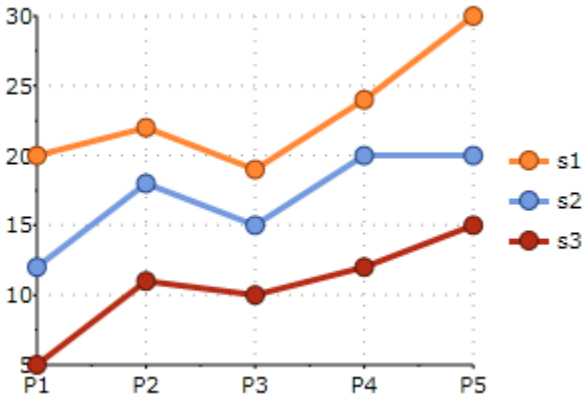

Module



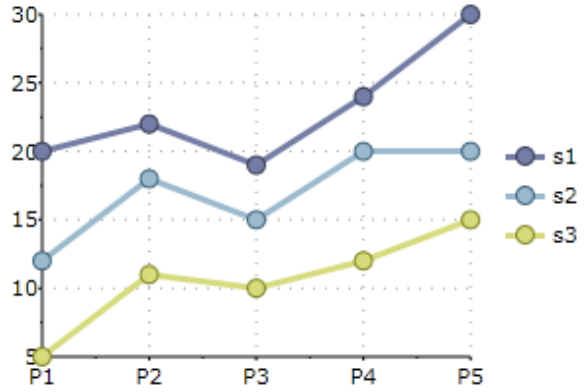

Opulent



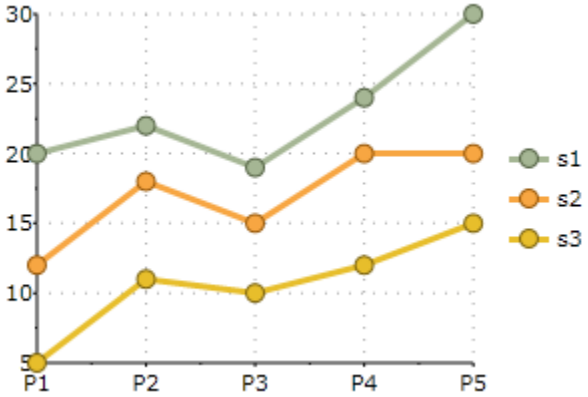
Oriel



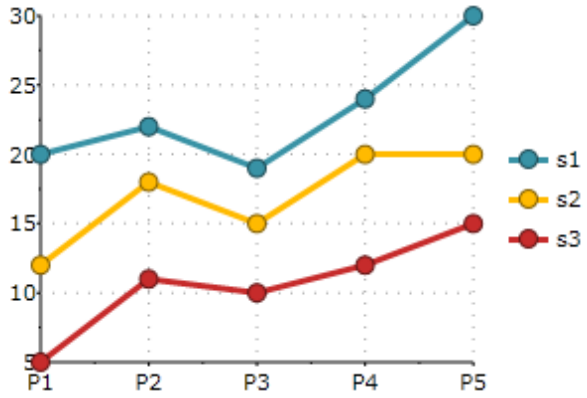

Origin



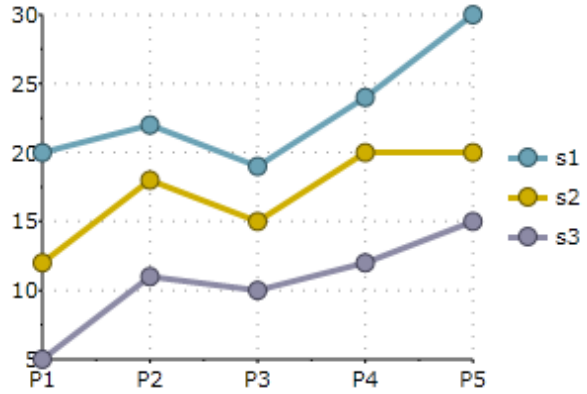

Paper



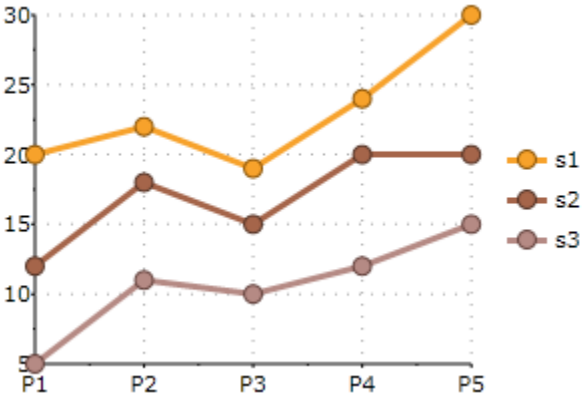
Solstice



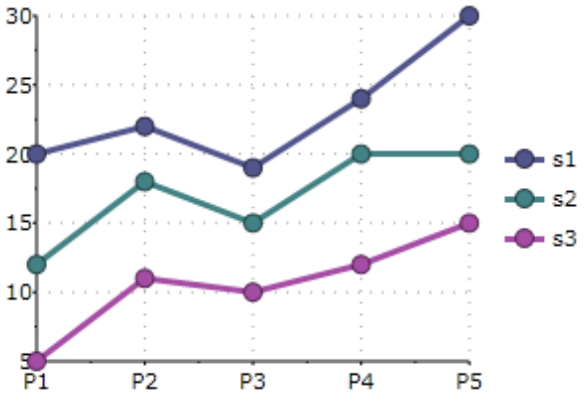

Technic



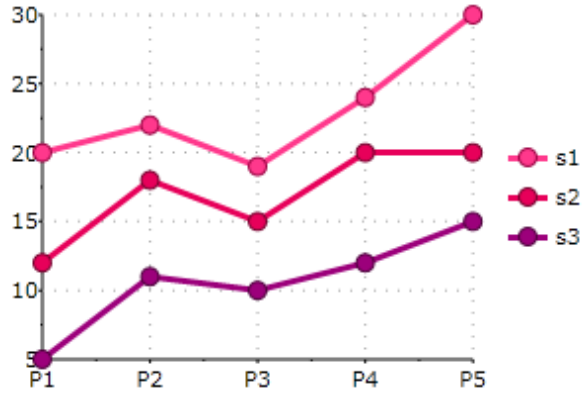

Trek



Urban




Verve




# Visual Effects

**Visual Effects** is a tool used for visually enhancing the 2D C1Chart control's border effect and shadow effect for the data series. Any existing project can use the new features provided by this tool. The chart's appearance can dramatically improve in a few simple steps using visual effects.

**Limitation**

Visual effects rendering may not work effectively in case of very complex and large data arrays or when highest performance is required.

C1Chart provides a VisualEffect class which is the base class. The following classes are derived from the VisualEffect class:

- BorderEffect – Defines the border effect for the data series. This class enables you to modify the border style, light direction, and thickness of the border.
- ShadowEffect – Defines the shadow effect for the data series. This class enables you modify the shadow color, shadow depth, light direction of the shadow, shadow opacity, and softness.
- VisualEffectGroup – Defines the visual effect that consists from the several effects.

## Border Effects

You can control the light angle, saturation, and transparency of the light source of the data series using the **BorderEffects** class. The **BorderEffects** class includes the follow properties:

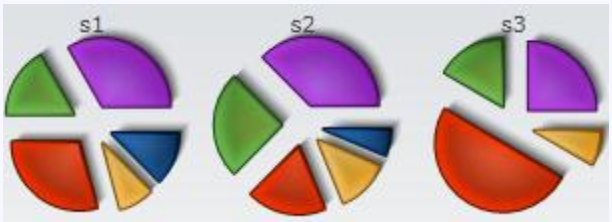
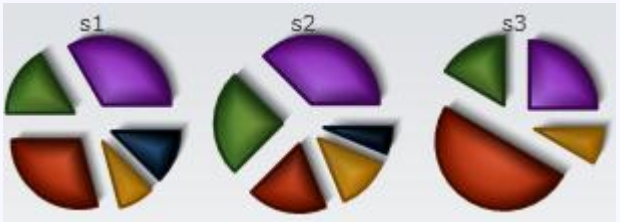
| Property | Description |
|---|---|

| | |
|---|---|
| BorderStyle | Gets or sets the border style. |
| LightAngle | Gets or sets the light direction. |
| Thickness | Gets or sets the thickness of the border. |

### BorderStyle Property

You can achieve a brightening or darkening effect on the data series using the BorderEffect.BorderStyle property. The BorderEffect.BorderStyle property provides three options: light, dark, and combo. The Light value will make the current data series color appear lighter in color. The Dark value produces a dark color effect for the current data series color. The Combo value produces a combination of the light and dark values for the current data series color.

The following table illustrates the effects of each value for the BorderEffect.BorderStyle property:

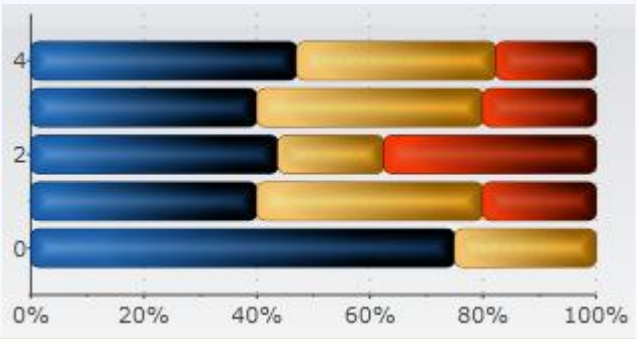
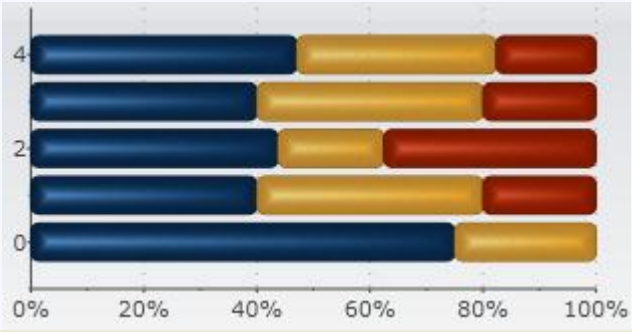
| Value | Result |
|---|---|
| **BorderStyle.Light** |  |
| **BorderStyle.Combo** |  |
| **BorderStyle.Dark** |  |

### LightAngle Property

You can change the angle of the visual rendering within the data series by setting its **LightAngle** property to a different degree. The value of the **LightAngle** property ranges from -180 to 180 degrees.

The following table illustrates the data series with a LightAngle value of 53 degrees and the data seires with a no LightAngle applied to it:

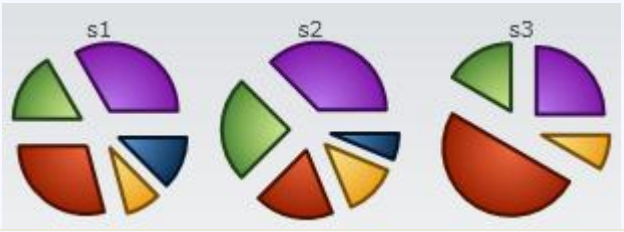
| Value | Result |
|---|---|

| | |
|---|---|
| 53 degree angle |  |
| None |  |

### Thickness Property

As you increase the **Thickness**, the light source within each series gradually decreases. When the Thickness property is set to a value higher than 25 the gradient light effects are not as noticeable.

The following table illustrates the effects of the various values for the BorderEffect.Thickness property:

| Numerical Value | Result |
|---|---|
| 0 |  |
| 10 |  |
| 25 |  |

## Shadow Effects

You can create shadow effects for the data series using the ShadowEffects class. The ShadowEffects class includes the following properties:

| Property | Description |
|---|---|
| Color | Gets or sets the shadow color. |
| Depth | Gets or sets the shadow depth. |
| LightAngle | Gets or sets the light direction. |
| Opacity | Gets or sets the shadow opacity (0-1). |
| Softness | Gets or sets the shadow softness. 0 is sharp shadow and 1 is soft shadow. |

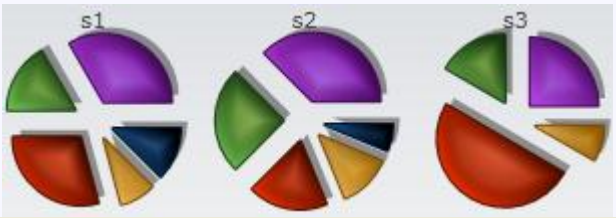
### Color Property

You can apply any color to the shadow using the Color property.

### Depth Property

You can use the Depth property to determine the depth of the shadow. Increasing the Depth value will make the shadow wider and appear further away from its associated object. Decreasing the Depth value will make the shadow narrower and appear closer to its associated object.

The following table illustrates the effects of the Depth property when set to different values.

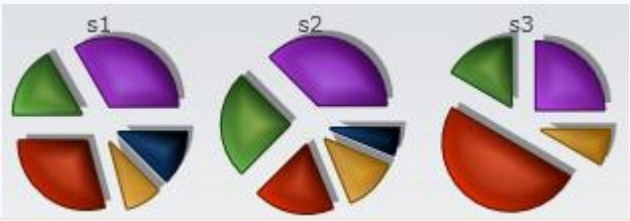
| Value | Result |
|---|---|
| 5 |  |
| 10 |  |

### LightAngle Property

You can change the angle of the visual rendering for the shadow by setting its **LightAngle** property to a different degree. The value of the **LightAngle** property ranges from -180 to 180 degrees.

The following table illustrates the data series with a LightAngle value of 44 degrees and the data seires with a no LightAngle applied to it:

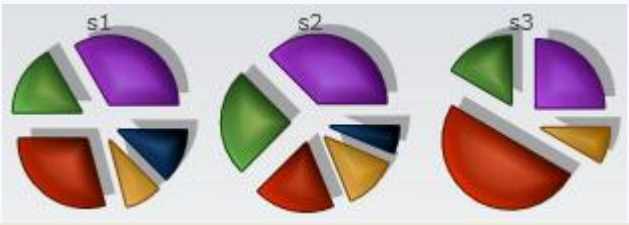
| Value | Result |
|---|---|

| | |
|---|---|
| 44 degrees |  |
| None |  |

**Opacity Property**

The Opacity property controls the opacity level for the shadow. You can darken the shadow around the data series by increasing value of the Opacity property. The numerical values range from 0.0 to 1.0. A value of 0 produces no color and a value of 1.0 produces a dark solid color.

The following table illustrates the data series with an **Opacity** value of 0.3 and the data series with an Opacity value of 1:

| Value | Result |
|---|---|
| 0.3 |  |
| 1.0 |  |

**Softness Property**

To make the shadow effects on the dataseries appear softer use the Softness property. A value of 0 will produce a sharp shadow that will make the shadow appear further from its associated object. A value of 1 will produce a soft shadow and make the shadow appear closer to its associated object.

The following table illustrates the data series with a Softness value of 0 and the data seires with a Softness value of 1:

| Value | Result |
|---|---|

| | |
|---|---|
| 0 |  |
| 1 |  |

# Chart for Silverlight 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.

> **Note:** ComponentOne samples are also available at http://helpcentral.componentone.com.

### C# ChartSamples Sample

| Sample | Description |
|---|---|
| **Advanced** | |
| AggregateChart | Represents built-in aggregate features. |
| Functions | Plots the function represented by javascript expression. |
| MovAverage | Creates two moving average trend lines with different periods. |
| Parametric | Plots the function defined parametrically with two equations x = x(t) and y = y(t). |
| PolygonPlot | Shows a Polygon chart with multiple series. |
| TrendLines | Shows available trend line types. |
| **Appearance** | |
| Animation | Shows animation when loading chart data. |
| Bubble | Shows a complex Bubble chart. |
| Effects | Built-in effects(border and shadow) applied to the plot elements. Demonstrates the effects of the following visual effects properties on any of the charts: BorderEffect.BorderStyle, BorderEffect.LightAngle, BorderEffect.Thickness, BorderStyle.Light, BorderStyle.Combo, BorderStyle.Dark, ShadowEffect.Color, ShadowEffect.Depth, ShadowEffect.LightAngle, ShadowEffect.Opacity, and ShadowEffect.Softness. |
| **Axes** | |

| | |
|---|---|
| AxisOrigin | Shows how to control the Axis position with its origin. |
| AxisTicks | Demonstrates various options for axis ticks. |
| CustomLabels | Shows how to customize the position and appearance of axis labels. |
| DependentAxes | Uses dependent axes to display the temperature in different units. |
| LogAxes | Shows various options of logarithmic chart axes. |
| **Basics** | |
| Gallery | Demonstrates the effect of the 12 different themes and palettes used on the various chart types. |
| ImageExport | Demonstrates chart export to image file. |
| Labels & Tooltips | Shows how to use labels and tooltips on Column, Line Symbols, and Pie charts. |
| Radar | Displays a Radar chart. |
| **Combination** | |
| ComplexChart | Illustrates a complex chart with different data series. |
| FinancialChart | Shows two y-axes for the Financial chart with column and candle data series. |
| PlotAreasHorizontal | Shows plot areas stacked horizontally. |
| PlotAreasVertical | Shows plot areas stacked vertically. |
| **Interaction** | |
| AddRemove | Shows how to add or remove markers. |
| DragDrop | Shows how to drag a column from a data series to another chart. |
| InteractiveChart | Demonstrates how to use the mouse button to zoom, the ctrl key and mouse button to scale, and the shift plus the mouse button to translate. It also shows how to zoom out. |
| Markers | Shows vertical and horizontal markers. |
| MouseMarker | Shows the marker moving with the mouse. |
| ZoomChart | Demonstrates built-in panning and zooming features. |
| **Performance** | |
| Dynamic | Shows dynamic data with trend lines. |
| LargeData | Interactive chart with 50000 data points. |

### C# ChartEditor Sample

| Sample | Description |
|---|---|
| ChartEditor | Demonstrates how to use ChartEditor control. Main properties of the chart can be edited at runtime with ChartEditor. The ChartEditor in this sample is docked at the right of the form. |

**C# ChartExtended Sample**

| Sample | Description |
|---|---|
| Functions | Plots simple function y=f(x). The code of function can be modified at runtime. |
| Parametric | Plots parametric function that is defined by two equations x=x(t) and y=y(t). |
| TrendLines | Shows various options of trend lines. |
| MovingAverage | Plots moving average trend. |
| Live | Plots dynamic data with automatic calculation of minimum, maximum and average. |

**To run any of the Chart for Silverlight samples:**

3. Open Visual Studio or Microsoft Expression Blend.
4. Select File | Open Project/Solution.
5. 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 Silverlight elsewhere on your machine.
6. Select the Samples.sln and click Open to open the project in Blend or Visual Studio
7. Select Project | Test Solution or click F5 to run the sample.

# Chart for Silverlight 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. If you are unfamiliar with the **ComponentOne Chart for Silverlight** product, please see the **Chart for Silverlight** Quick Start first.

Each topic in this section provides a solution for specific tasks using the **ComponentOne Chart for Silverlight** product.

Each task-based help topic also assumes that you have created a new Silverlight project.

## Axes Tasks

The following topics show how to modify the axes origin, tick marks, and annotation.

### Inheriting Data from the Parent Chart

The axes can inherit data from the parent chart by using the following code:

```
<c1:C1Chart Name="chart" ChartType="Column" >
<c1:C1Chart.Data>
<c1:ChartData>
<c1:DataSeries SymbolFill="{Binding Brush1}" Values="1,2,3,4" />
<c1:DataSeries SymbolFill="{Binding Brush2}" Values="4,3,2,1" />
</c1:ChartData>
</c1:C1Chart.Data>
```

```
<c1:C1Chart.View>
<c1:ChartView>
<c1:ChartView.AxisX>
<c1:Axis Max="{Binding XMax}" />
</c1:ChartView.AxisX>
<c1:ChartView.AxisY>
<c1:Axis Max="{Binding YMax}" />
</c1:ChartView.AxisY>
</c1:ChartView>
</c1:C1Chart.View>
</c1:C1Chart>

where data context is

public class ChartDataContext
{
public Brush Brush1 { get; set; }
public Brush Brush2 { get; set; }

public double XMax { get; set; }
public double YMax { get; set; }
}

// set data context
chart.DataContext = new ChartDataContext()
{
Brush1 = new SolidColorBrush(Colors.Red),
Brush2 = new SolidColorBrush(Colors.Blue),
XMax = 10,
YMax = 10
};
```

### Adding Auxiliary for Second Data Point

To add auxiliary axis for second(index=1) data point, use the following code:

```
chart.View.Axes.Add(new Axis() { AxisType= AxisType.Y,
RadarPointIndices=new int[] {1} });
```

### Customizing Axis Labels using Axis.AnnoCreated Event

To set the colors of the labels depending upon its values, use the following code:

```
chart.View.AxisY.AnnoCreated += (s, e) =>
{
var label = (TextBlock)e.Label;
if (e.Value >= 0)
label.Foreground = Brushes.Red;
else
label.Foreground = Brushes.Blue;
};
```

### 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;
   }
}
```

### Creating a Custom Axis Label to Set the Mark at the Middle of the Year

Use custom axis labels to set the mark at the middle of year like in the following code:

- Visual Basic

```
chart.View.AxisX.ItemsSource = New DateTime() {New DateTime(2006, 1,
1).AddDays(0.5 * 365), New DateTime(2007, 1, 1).AddDays(0.5 * 365)}
```

- C#

```
chart.View.AxisX.ItemsSource = new DateTime[]
   {
     new DateTime(2006,1,1).AddDays(0.5*365),
     new DateTime(2007,1,1).AddDays(0.5*365),
     //...
   };
```

## Placing the Chart Annotations on Top

To place the chart annotations on top, use the following code:

- Visual Basic

```
chart.View.AxisX.Position = AxisPosition.Far
```

- C#

```
chart.View.AxisX.Position = AxisPosition.Far;
```

## Rotating the Chart Labels in the X-Axis

To change the label rotation for the X-Axis, use the **AnnoAngle** property like the following:

```
AxisX.AnnoAngle = -90
```

## 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

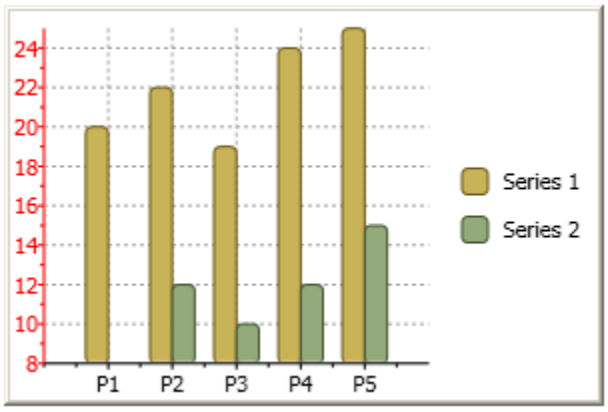
There are two types of ticks on the axis: major tick has a small line and corresponding label while the minor tick has only the line across the axis.

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 Tasks

The following topics show how to modify the chart's data labels, plot element colors, and chart legend.

## Adding a ScrollBar to C1Chart

You can add a scrollbar to C1Chart by linking the chart axis with the standard scrollbar like the following:

- XAML

```
<Grid x:Name="LayoutRoot">
    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition Height="Auto" />
```

```
    </Grid.RowDefinitions>
    <c1chart:C1Chart x:Name="chart" ></c1chart:C1Chart>
    <ScrollBar x:Name="sb" Orientation="Horizontal" Grid.Row="1" />
</Grid>
```

- C#

```
// create some data
      int npts = 1000;
      double[] x = new double[npts], y = new double[npts];
      for (int i = 0; i < npts; i++)
      {
        x[i] = i; y[i] = 100 * Math.Sin(0.1 * i) * Math.Cos(0.01 * i);
      }

      // add data to the chart
      chart.Data.Children.Add(new XYDataSeries() { XValuesSource = x,
ValuesSource = y });
      chart.ChartType = ChartType.Line;

      // setup axis
      double xscale = 0.05; // show only 1/20 of the full data range
      chart.View.AxisX.Min = 0; chart.View.AxisX.Max = npts-1;
      chart.View.AxisX.Scale = xscale;

      sb.Minimum = 0; sb.Maximum = 1;
      sb.SmallChange = 0.5*xscale; sb.LargeChange = xscale;
      sb.ViewportSize = 1.0 / (1.0 - xscale) - 1.0;
      // connect axis with toolbar
      sb.ValueChanged += (s, e) => chart.View.AxisX.Value = sb.Value;
```

## Breaking Chart Lines when the Y-Value is Null

To break the chart lines when the Y-Value is null, use the following code:

- Visual Basic

```
dataSeries.Display = SeriesDisplay.ShowNaNGap
```

- C#

```
dataSeries.Display = SeriesDisplay.ShowNaNGap;
```

## 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;
```

## Creating Wrap Around Text Blocks for Large Number of Series

To create wrap around text blocks for large number of series use the following XAML and then use the Visual Basic or C# code:

- XAML

```
<c1chart:C1Chart x:Name="chart">
```

```
      <c1chart:C1ChartLegend Position="Top">
        <c1chart:C1ChartLegend.ItemsPanel>
          <ItemsPanelTemplate>
            <c1:C1WrapPanel ItemHeight="20" ItemWidth="80"/>
          </ItemsPanelTemplate>
        </c1chart:C1ChartLegend.ItemsPanel>
      </c1chart:C1ChartLegend>
</c1chart:C1Chart>
```

- Visual Basic

```
For i As Integer = 0 To 19
    chart.Data.Children.Add(New DataSeries())
Next
```

- C#

```
for (int i = 0; i < 20; i++)
   chart.Data.Children.Add(new DataSeries() { Label = "s " + i.ToString()
});
```

## Displaying Gaps in Line or Area Charts

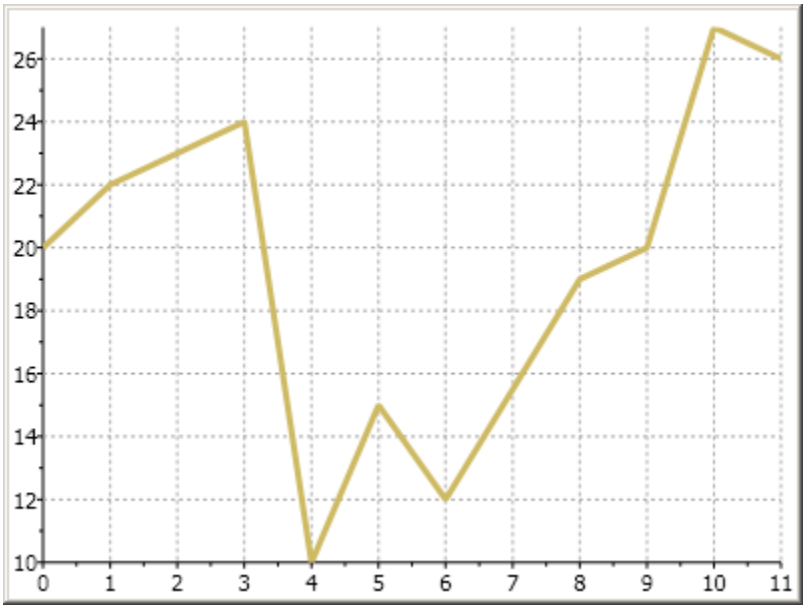
By default, if there is a hole(double.NaN) in data values chart just skips the value and draws line to next valid data point.

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:

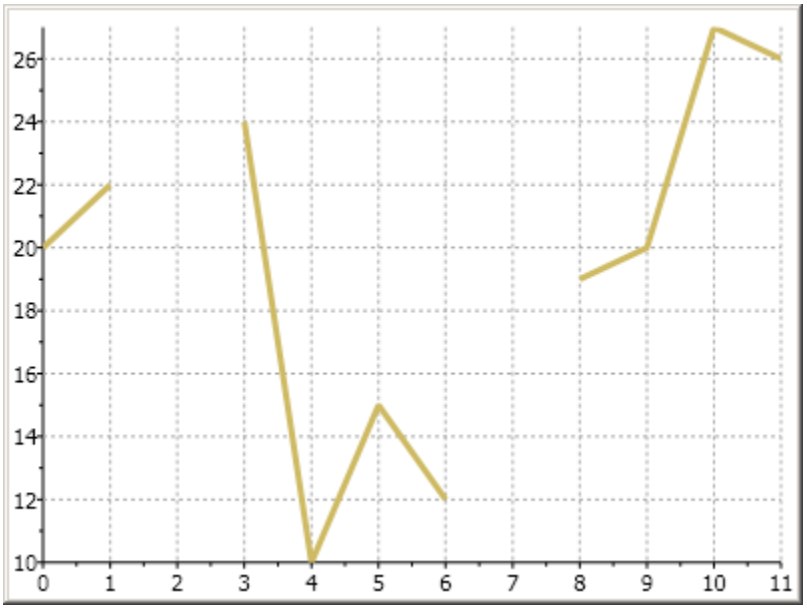
- Visual Basic

```
Me.C1Chart1.Data.Children(1).Display =
C1.Silverlight.C1Chart.SeriesDisplay.ShowNaNGap
```

- C#

```
this.C1Chart1.Data.Children[1].Display =
C1.Silverlight.C1Chart.SeriesDisplay.ShowNaNGap;
```

The line chart will display a gap between the chart lines, similar to the following:




## Finding the Default Color Used for the Series

When defining a series programmatically you can find out what series color will be drawn in using the Loaded event, like the following:

```
DataSeries ds = new DataSeries() { ValuesSource = new double[] { 1, 2,
3 } };
ds.Loaded += (s, e) => // fired for each plot element in data series
{
    PlotElement pe = (PlotElement)s;
    SolidColorBrush sb = pe.Fill as SolidColorBrush;
    if( sb!=null)
        Debug.WriteLine(sb.Color);
};

chart.Data.Children.Add(ds);
```

## 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
```

## Setting Custom Colors in DataSeries.Loaded Event

You can set different colors for the data point in the same series using the Loaded event like the following:

```
DataSeries ds = new DataSeries()
```

```
 {
   ValuesSource = new double[] {-2, -1, 1, 2, 2, 1, -1, -3 }
 };

SolidColorBrush red = new SolidColorBrush(Colors.Red);
SolidColorBrush blue = new SolidColorBrush(Colors.Blue);

ds.Loaded += (s, e) =>
   {
     PlotElement pe = s as PlotElement;
     if (pe != null &&
       pe.DataPoint.PointIndex >= 0) // only points
     {
       // set color depending on point value
       if (pe.DataPoint.Value >= 0)
         pe.Fill = red;
       else
         pe.Fill = blue;
     }
   };

chart.Data.Children.Add(ds);
chart.ChartType = ChartType.LineSymbols;
```

## 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;
```

## Showing the X-Values in the Tooltip

To show the x-values in the tooltip, use the following xaml and then the Visual Basic or C# code that creates the chart and uses the template:

- XAML

```
<c1chart:C1Chart x:Name="chart">
   <c1chart:C1Chart.Resources>
      <DataTemplate x:Key="tt">
        <StackPanel Orientation="Vertical">
          <!-- XAsString returns xvalue that is formatted similar to x-
axis -->
          <TextBlock Text="{Binding XAsString}" />
          <TextBlock Text="{Binding Y}" />
```

```
        </StackPanel>
      </DataTemplate>
    </c1chart:C1Chart.Resources>
</c1chart:C1Chart>
```

- Visual Basic

```
Dim cnt As Integer = 20
Dim x As DateTime() = New DateTime(cnt - 1) {}
Dim y As Double() = New Double(cnt - 1) {}
Dim rnd As New Random()
For i As Integer = 0 To cnt - 1
    x(i) = DateTime.Today.AddDays(-i)
    y(i) = rnd.NextDouble() * 100
Next

chart.Data.Children.Add(New XYDataSeries())

chart.View.AxisX.IsTime = True
chart.ChartType = ChartType.LineSymbols
```

- C#

```
int cnt = 20;
  DateTime[] x = new DateTime[cnt];
  double[] y = new double[cnt];
  Random rnd = new Random();
  for (int i = 0; i < cnt; i++)
  {
     x[i] = DateTime.Today.AddDays(-i);
     y[i] = rnd.NextDouble() * 100;
  }

  chart.Data.Children.Add(new XYDataSeries()
  {
     XValuesSource = x, ValuesSource = y,
     PointTooltipTemplate = (DataTemplate)chart.Resources["tt"]
  });

  chart.View.AxisX.IsTime = true;
  chart.ChartType = ChartType.LineSymbols;
```

## Showing Trend Marks in C1Chart

The following example code shows how to use the trend indicators in the chart with triangles used as arrows:

- Visual Basic

```
chart.ChartType = ChartType.LineSymbols

Dim values As Double() = New Double() {1, 4, 3, 2, 4, 7}

Dim ds As New DataSeries()

ds.Loaded += Function(sender, args) Do
    Dim rp As RPolygon = TryCast(sender, RPolygon)

    If rp IsNot Nothing Then
```

```
        Dim pi As Integer = rp.DataPoint.PointIndex
        If pi > 0 Then
            'rotate triangle and change its color
            If values(pi) > values(pi - 1) Then
                rp.RenderTransform = New RotateTransform()
                rp.RenderTransformOrigin = New Point(0.5, 0.5)
                rp.Fill = New SolidColorBrush(Colors.Green)
            ElseIf values(pi) < values(pi - 1) Then
                rp.RenderTransform = New RotateTransform()
                rp.RenderTransformOrigin = New Point(0.5, 0.5)
                rp.Fill = New SolidColorBrush(Colors.Red)
            End If
        End If
    End If
End Function

chart.Data.Children.Add(ds)
```

- C#

```
chart.ChartType = ChartType.LineSymbols;

double[] values = new double[] { 1, 4, 3, 2, 4, 7 };

DataSeries ds = new DataSeries()
{
  ConnectionStroke = new SolidColorBrush(Colors.DarkGray),
  ConnectionStrokeThickness=1,
  Symbol = new RPolygon(),
  SymbolSize = new Size(18,12),
  ValuesSource = values
};

ds.Loaded += (sender, args) =>
  {
    RPolygon rp = sender as RPolygon;

    if (rp != null)
    {
      int pi = rp.DataPoint.PointIndex;
      if (pi > 0)
      {
        //rotate triangle and change its color
        if ( values[pi] > values[pi-1] )
        {
          rp.RenderTransform = new RotateTransform() { Angle = -90 };
          rp.RenderTransformOrigin = new Point(0.5, 0.5);
          rp.Fill = new SolidColorBrush(Colors.Green);
        }
        else if (values[pi] < values[pi - 1])
        {
          rp.RenderTransform = new RotateTransform() { Angle = 90 };
          rp.RenderTransformOrigin = new Point(0.5, 0.5);
          rp.Fill = new SolidColorBrush(Colors.Red);
        }
      }
    }
  };
```

```
chart.Data.Children.Add(ds);
```

### 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

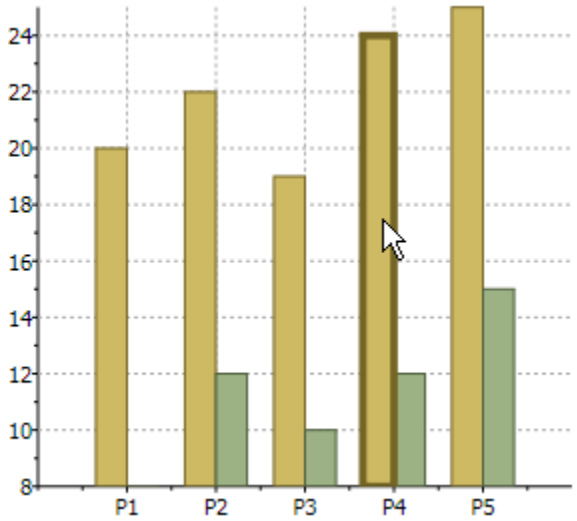
You can add animation when you click on any column in the Column chart, using **MouseDown** and **MouseLeave** events, like the following XAML code:

```
<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.

### Creating a Mouse Move Event for Chart with Multiple Lines

You can use the **PlotElement.MouseEnter** event to create a mouse move event for chart with multiple lines like the following:

```
Lines lines = new Lines() { StrokeThickness = 4 };
lines.MouseEnter += (s, e) =>
{
   // fires when mouse is over the lines
   Debug.WriteLine("MouseEnter");
   // show coordinates
   ...
};
dataSeries.Connection = lines;
```

### Displaying DataSeries Label as Tooltip on Mouseover in Line Chart

To display DataSeries label as a tooltip on mouseover in a Line chart, use the following code:

```
DataSeries ds = new DataSeries()
{
   ValuesSource = new double[] { 1, 2, 3, 2 },
   ConnectionStrokeThickness = 5
};
ds.Loaded += (s, e) =>
{
  PlotElement pe = s as PlotElement;
  if (pe != null)
    pe.MouseEnter += (sender, ea) =>
```

```
    {
      Debug.WriteLine("Series index = {0}", pe.DataPoint.SeriesIndex);
    };
};

chart.Data.Children.Add(ds);
chart.ChartType = ChartType.Line;
```

### 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 and/or modify different chart types.

## Adding XYDataSeries to a Stacked Column Chart

To add XYDataSeries to a **Stacked Column** chart, complete the following:

```
// 5 points
var ds1 = new XYDataSeries()
{
XValuesSource = new double[]{ 1, 2, 3, 4, 5},
ValuesSource = new double[] { 1, 1, 1, 1, 1}
};
chart.Data.Children.Add(ds1);

// 3 points
var ds2 = new XYDataSeries()
{
XValuesSource = new double[] { 1, 3, 5 },
ValuesSource = new double[] { 1, 1, 1 }
};
chart.Data.Children.Add(ds2);

chart.ChartType = ChartType.ColumnStacked;
```

## 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" />
```

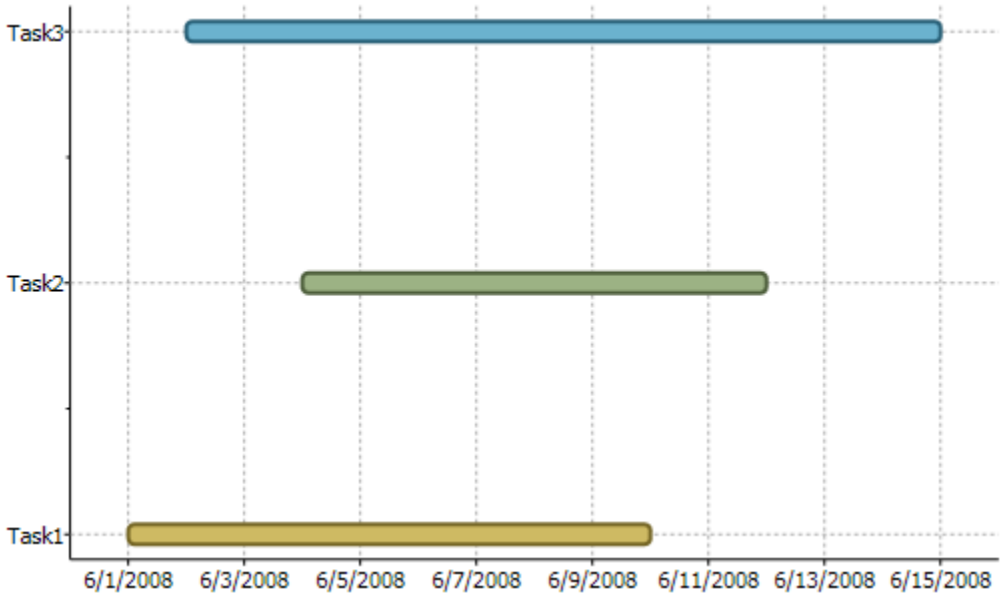
```
     </c1chart:ChartData>
   </c1chart:C1Chart.Data>
</c1chart:C1Chart>
```

### 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 Data Values to a Pie Chart

The following code adds data values to a Pie chart:

- Visual Basic

```
Dim names As String() = New String() {"Apples", "Oranges", "Peaches"}
Dim values As Double() = New Double() {20R, 10R, 7R}

Dim ds As New DataSeries()
```

```
chart.Data.Children.Add(ds)

chart.Data.ItemNames = names
chart.ChartType = ChartType.Pie
```

- C#

```
string[] names = new string[] { "Apples", "Oranges", "Peaches"};
double[] values = new double[] { 20.0, 10.0, 7.0 };

DataSeries ds = new DataSeries() { ValuesSource = values};
chart.Data.Children.Add(ds);

chart.Data.ItemNames = names;
chart.ChartType = ChartType.Pie;
```

## 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 Individual Offset for Each Pie Slice

To set the offset for each pie slice, use the following code:

```
var ds = new DataSeries() { ValuesSource = new double[] { 1, 2, 2, 1, 2, 1, 2
} };
ds.PlotElementLoaded += (s, e) =>
{
var slice = s as PieSlice;
if (slice != null)
{
// add offset for even slices
if ((slice.DataPoint.PointIndex % 2) == 0)
slice.Offset = 15;
}
};

chart.Data.Children.Add(ds);
chart.ChartType = ChartType.Pie;
```

### 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);
```

## 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);
```

### Exporting the Chart Image to the PDF on the Client

To export the chart image to the PDF on the client, use the following code:

- Visual Basic

```
Private Sub Button_Click(ByVal sender As Object, ByVal e As
RoutedEventArgs)
    Dim wb As New WriteableBitmap(chart, Nothing)
    Dim doc As New C1PdfDocument()
    doc.Compression = CompressionLevelEnum.BestSpeed
    doc.DrawImage(wb, New Rect(New Point(), doc.PageSize),
ContentAlignment.TopLeft, Stretch.None)

    Dim sfd As New SaveFileDialog()
    If sfd.ShowDialog() = True Then
        Using stream = sfd.OpenFile()
            doc.Save(stream)
        End Using
    End If
End Sub
```

- C#

```
private void Button_Click(object sender, RoutedEventArgs e)
{
   WriteableBitmap wb = new WriteableBitmap(chart, null);
   C1PdfDocument doc = new C1PdfDocument();
   doc.Compression = CompressionLevelEnum.BestSpeed;
   doc.DrawImage(wb, new Rect(new Point(),doc.PageSize),
ContentAlignment.TopLeft, Stretch.None);

   SaveFileDialog sfd = new SaveFileDialog()
   {
      DefaultExt = "pdf",
      Filter = "Pdf files (*.pdf)|*.pdf",
   };
   if (sfd.ShowDialog() == true)
   {
     using (var stream = sfd.OpenFile())
     {
        doc.Save(stream);
     }
   }
}
```

### 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.Silverlight.C1Chart.Extended.ImageFormat.Png)
End Using
```

- C#

```
// save image to file
      using (var stm = System.IO.File.Create("chart.png"))
      {
        c1Chart1.SaveImage(stm,
C1.Silverlight.C1Chart.Extended.ImageFormat.Png);
      }
```

### Saving the Chart Image as Jpeg or Png

To save the C1Chart control as a .Jpg or .Png, use the following code:

```
// should be added at the beginning of file
    using C1.Silverlight.Chart.Extended;
   ...

    // Silverlight
    SaveFileDialog sfd = new SaveFileDialog();
    sfd.Filter = "Jpeg files (*.jpg)|*.jpg|Png files (*.png)|*.png";
    if (sfd.ShowDialog() == true)
    {
      using (var stream = sfd.OpenFile())
      {
       if (sfd.SafeFileName.EndsWith(".jpg"))
         chart.SaveImage(stream, ImageFormat.Jpeg);
       else
       chart.SaveImage(stream, ImageFormat.Png);
      }
    }
```

# Converting DataSet to DataSeries

You can convert the DataSet to DataSeries using just one DataSeries like the following:

- XAML

```
C1Chart1.Theme = ChartTheme.Office2003Classic
```

- Visual Basic

```
Dim dt As New DataTable()
dt.Columns.Add("Name", GetType(String))
dt.Columns.Add("Value", GetType(Double))
dt.Rows.Add("Apples", 20R)
dt.Rows.Add("Oranges", 10R)
dt.Rows.Add("Peaches", 7R)

chart.Data.ItemNameBinding = New Binding("Name")
chart.Data.Children.Add(New DataSeries())
chart.Data.ItemsSource = dt.DefaultView
chart.ChartType = ChartType.Pie
```

- C#

```
DataTable dt = new DataTable();
dt.Columns.Add("Name", typeof(string));
dt.Columns.Add("Value", typeof(double));
dt.Rows.Add("Apples", 20.0);
dt.Rows.Add("Oranges", 10.0);
dt.Rows.Add("Peaches", 7.0);

chart.Data.ItemNameBinding = new Binding("Name");
chart.Data.Children.Add(new DataSeries() { ValueBinding = new
Binding("Value") });
chart.Data.ItemsSource = dt.DefaultView;
chart.ChartType = ChartType.Pie;
```