

---

ComponentOne

# Chart for Silverlight

Copyright © 1987-2009 ComponentOne LLC. All rights reserved.

*Corporate Headquarters*  
**ComponentOne LLC**  
201 South Highland Avenue  
3<sup>rd</sup> Floor  
Pittsburgh, PA 15206 • USA

**Internet:** [info@ComponentOne.com](mailto:info@ComponentOne.com)  
**Web site:** <http://www.componentone.com>

**Sales**

E-mail: [sales@componentone.com](mailto:sales@componentone.com)  
Telephone: 1.800.858.2739 or 1.412.681.4343 (Pittsburgh, PA USA Office)

**Trademarks**

ComponentOne Chart for Silverlight and the ComponentOne Chart for Silverlight logo are trademarks of ComponentOne LLC. ComponentOne is a registered trademark of ComponentOne LLC. All other trademarks used herein are the properties of their respective owners.

**Warranty**

ComponentOne warrants that the original media is free from defects in material and workmanship, assuming normal use, for a period of 90 days from the date of purchase. If a defect occurs during this time, you may return the defective media to ComponentOne, along with a dated proof of purchase, and ComponentOne will replace it at no cost to you. After 90 days, you may obtain a replacement for a defective media by sending your request and a check for \$25 (to cover postage and handling) to ComponentOne at the above address.

Except for the express warranty of the original media set forth herein, ComponentOne makes no other warranties, express or implied. Every attempt has been made to ensure that the information contained in this manual is correct as of the time it was made public. ComponentOne does not warrant and therefore shall not be liable for any errors or omissions that the software documentation may contain. ComponentOne's liability is limited to the amount you paid for the product. ComponentOne is not and shall not be held liable for any special, punitive, incidental, consequential, or any other damages that may result from your use of the software.

**Copying and Distribution**

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



# Table of Contents

<b>ComponentOne Chart for Silverlight</b> .....	<b>1</b>
Introduction to C1.Silverlight.Chart .....	1
C1Chart Concepts and Main Properties .....	1
Basic Charting .....	2
Simple Charts.....	2
Time-Series Charts .....	6
XY Charts.....	9
Data-Binding .....	12
Data-Binding to C1.Silverlight.Data.....	14
Data Labels and Tooltips.....	16
<b>Advanced Topics</b> .....	<b>21</b>
Animation .....	21
OnLoad Animations.....	21
OnMouseOver Animations.....	22
Zooming and Panning .....	24
Zooming and Panning with Two Charts.....	24
Zooming and Panning with a Single Chart .....	27
Attaching Elements to Data Points .....	28
<b>Specialized Charts</b> .....	<b>31</b>
Financial Charts .....	31
Gantt Charts.....	32
<b>Using XAML</b> .....	<b>37</b>



# ComponentOne Chart for Silverlight

The **C1.Silverlight.Chart** assembly contains two main objects: the **C1Chart** and **C1ChartLegend** controls.

**C1Chart** is a powerful charting tool that allows you to display quantitative data in clear and attractive ways. **C1Chart** also supports advanced features such as animations and user interactions, including zooming and panning.

**C1ChartLegend** is a separate control used to display a chart legend for a given **C1Chart** control. The legend is implemented as a separate control to leverage the Silverlight/WPF layout mechanism.

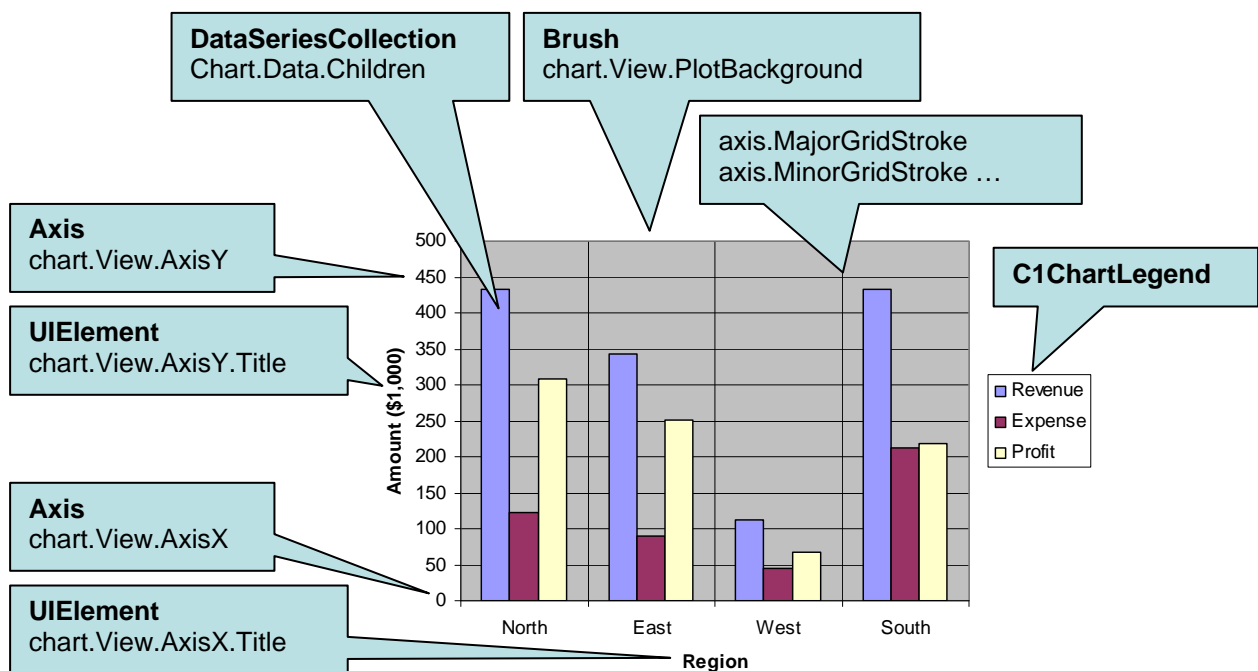
## Introduction to C1.Silverlight.Chart

This document introduces the **C1Chart** control by discussing basic concepts and showing how you can use the control to create attractive charts quickly and easily. Advanced topics such as animation and interaction are also covered in later chapters.

The document includes code to illustrate the concepts, but it does not follow a tutorial format. The application described here is called "ChartIntro"; it can be found in the samples folder installed with the ComponentOne Studio for Silverlight product.

## C1Chart Concepts and Main Properties

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



The steps involved in creating a typical chart are:

1. Choose the chart type (ChartType property)  
**CIChart** supports about 30 chart types, including Bar, Column, Line, Area, Pie, Radial, Polar, Candle, and several others. The best chart type depends largely on the nature of the data, and will be discussed later.
2. Set up the axes (AxisX and AxisY properties)  
Setting up the axes typically involves specifying the axis title, major and minor intervals for the tick marks, content and format for the labels to show next to the tick marks.
3. Add one or more data series (Children collection)  
This step involves creating and populating one `DataSeries` object for each series on the chart, then adding the object to the `Children` collection. If your data contains only one numeric value per point (Y coordinate), use regular `DataSeries` objects. If the data contains two numeric values per point (X and Y coordinates), then use `XYDataSeries` objects instead.
4. Adjust the chart's appearance using the Theme and Palette properties.  
The Theme property allows you to select one of over 10 built-in sets of properties that control the appearance of the overall chart. The Palette property allows you to select one of over 20 built-in color palettes used to specify colors for the data series. Together, these two properties provide about 200 options to create professionally-looking charts with little effort.

The next sections will walk you through some actual code.

## Basic Charting

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 one shown in the diagram above.

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 the following code, this is just some generic data.

```
// 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.

Now that we have the data, let's walk through a method that creates simple charts. The method is called `BuildSalesPerRegionChart`; it assumes you have a **CIChart** control on the page, named `_c1Chart`, and takes the chart type as a parameter.

We will follow the four steps outlined above.

**Step 1) Choose the chart type:**

In this case, the chart type is one of the parameters of the method. The code clears any existing series, then sets the chart type:

```
void BuildSalesPerRegionChart(ChartType chartType)
{
    // clear current chart
    _clChart.Reset(true);

    // set chart type
    _clChart.ChartType = chartType;
}
```

**Step 2) Set up the axes:**

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

```
// get axes
Axis valueAxis = _clChart.View.AxisY;
Axis labelAxis = _clChart.View.AxisX;
if (chartType == ChartType.Bar)
{
    valueAxis = _clChart.View.AxisX;
    labelAxis = _clChart.View.AxisY;
}
```

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
valueAxis.Title = CreateTextBlock("Amount ($1000)", 14,
FontWeights.Bold);
valueAxis.AutoMin = false;
valueAxis.Min = 0;
valueAxis.MajorUnit = 200;
valueAxis.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
_clChart.Data.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();
_clChart.Data.Children.Add(ds);
// add Expense series
ds = new DataSeries();
ds.Label = "Expense";
ds.ValuesSource = (from r in data select r.Expense).ToArray();
_clChart.Data.Children.Add(ds);
// add Profit series
ds = new DataSeries();
ds.Label = "Profit";
ds.ValuesSource = (from r in data select r.Profit).ToArray();
_clChart.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` and `Palette` properties to quickly configure the chart appearance:

```
// set theme and palette
_clChart.Theme = ChartTheme.Office2007Black;
_clChart.Palette = Palette.Default;
}
```

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

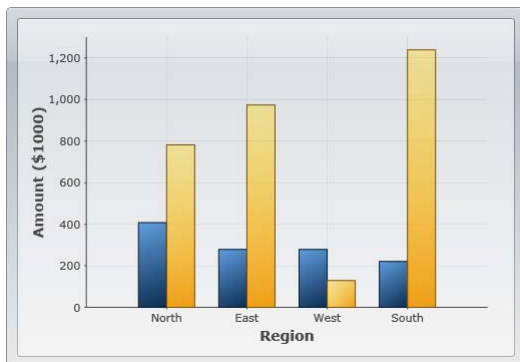
```
TextBlock CreateTextBlock(string text, double fontSize, FontWeight
fontWeight)
{
```

```

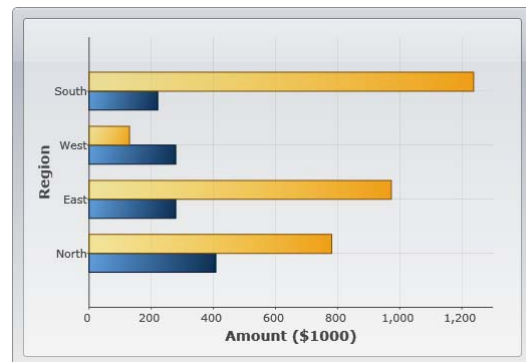
var tb = new TextBlock();
tb.Text = text;
tb.FontSize = fontSize;
tb.FontWeight = fontWeight;
return tb;
}

```

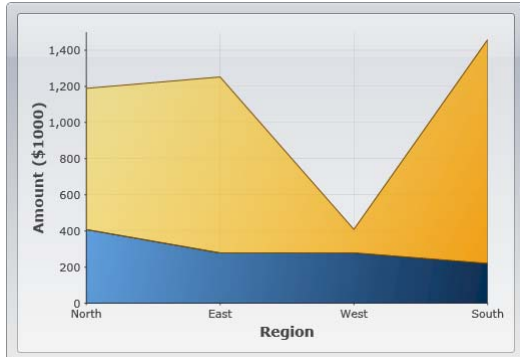
This concludes the code that generates simple value charts. You can test it by invoking the **BuildSalesPerRegionChart** to create charts of different types. The result should be similar to the images below:



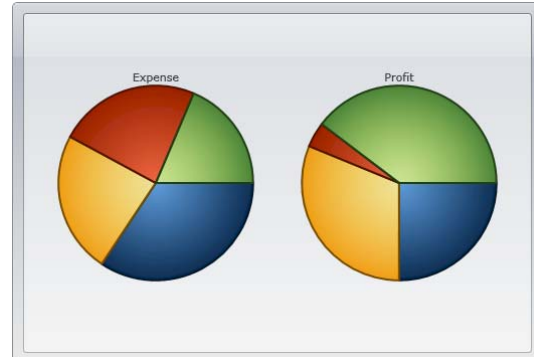
BuildSalesPerRegionChart(  
**ChartType.Column**)



BuildSalesPerRegionChart(  
**ChartType.Bar**)



BuildSalesPerRegionChart(  
**ChartType.AreaStacked**)



BuildSalesPerRegionChart(  
**ChartType.Pie**)

Note that the chart does not display a legend describing the series. To do that, we would use a separate control, the **C1ChartLegend**, which would be positioned on the page independently of the chart. Linking a **C1ChartLegend** control to a **C1Chart** is simple, all you need to do is set one property:

```

// link chart legend to the chart
_c1ChartLegend.Chart = _c1Chart;

```

## 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. The code is similar to what we used when creating the simple charts in the previous section.

### Step 1) Choose the chart type:

As before, the chart type is one of the parameters of the method. The code clears any existing series, then sets the chart type:

```
void BuildSalesPerMonthChart(ChartType chartType)
{
    // clear current chart
    _clChart.Reset(true);

    // set chart type
    _clChart.ChartType = chartType;
```

### 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 = _clChart.View.AxisY;
Axis labelAxis = _clChart.View.AxisX;
if (chartType == ChartType.Bar)
{
    valueAxis = _clChart.View.AxisX;
    labelAxis = _clChart.View.AxisY;
}
```

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

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

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

### Step 3) Add one or more data series

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

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

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

```
// show regions along label axis
_clChart.Data.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();
    _clChart.Data.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.

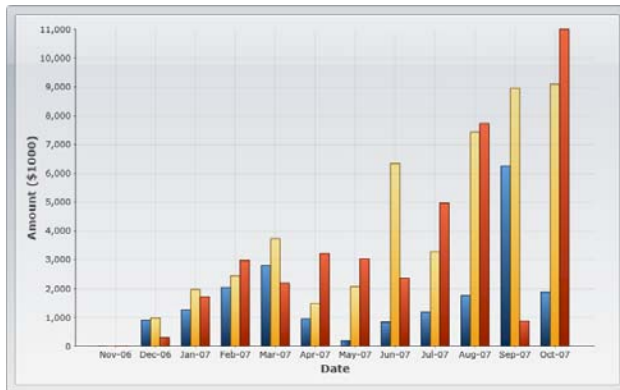
#### Step 4) Adjust the chart's appearance

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

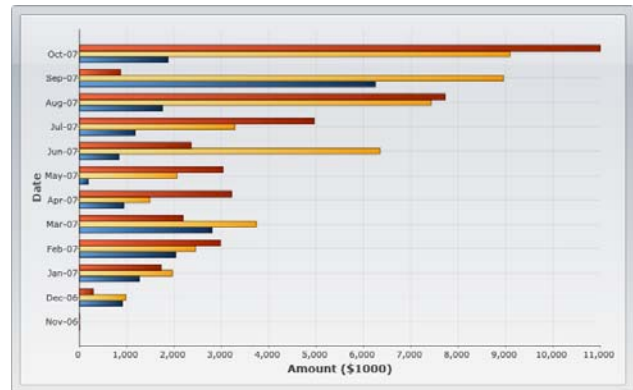
```
// set theme and palette
_clChart.Theme = ChartTheme.Office2007Black;
_clChart.Palette = Palette.Default;
}
```

---

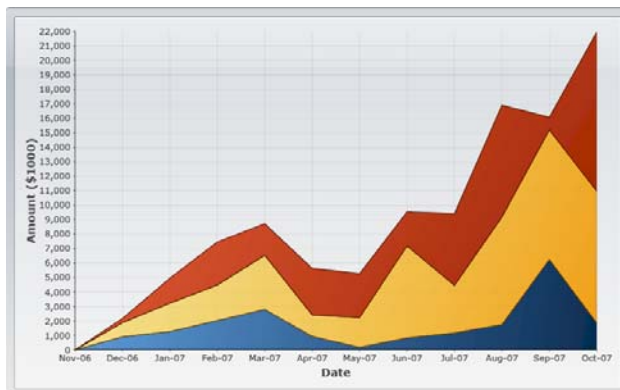
This concludes the code that generates our time-series charts. You can test it by invoking the **BuildSalesPerMonthChart** to create charts of different types. The result should be similar to the images below:



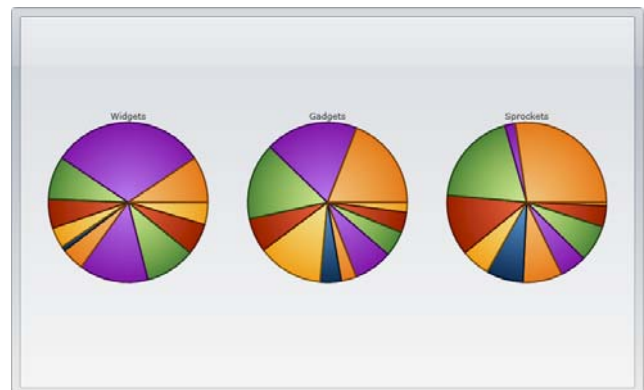
BuildSalesPerMonthChart (  
ChartType.Column)



BuildSalesPerMonthChart (  
ChartType.Bar)



BuildSalesPerMonthChart (  
ChartType.AreaStacked)



BuildSalesPerMonthChart (  
ChartType.Pie)

As before, the chart does not display a legend describing the series. To do that, we would use a separate control, the **C1ChartLegend**, which would be positioned on the page independently of the chart.

Also, 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 `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:

As before, the chart type is passed as a parameter. The code clears any existing series, then sets the chart type:

```
void BuildRevenueCorrelationChart(ChartType chartType)
{
    // clear current chart
    _clChart.Reset(true);

    // set chart type
    _clChart.ChartType = chartType;
}
```

### 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 = _clChart.View.AxisY;
var xAxis = _clChart.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.Data.Children.Add(ds);

```

#### 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:

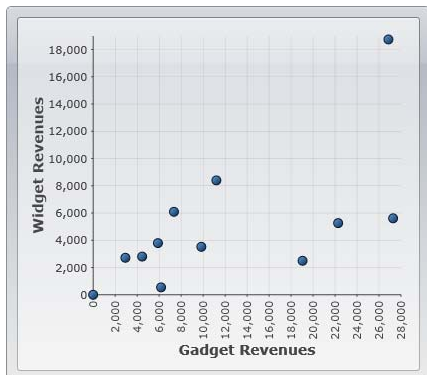
```

// set theme and palette
_c1Chart.Theme = ChartTheme.Office2007Black;
_c1Chart.Palette = Palette.Default;
}

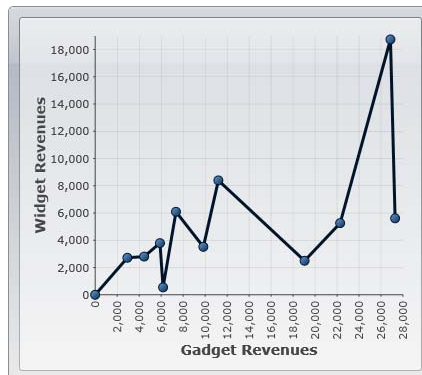
```

This concludes the code that generates our XY charts. You can test it by invoking the **BuildRevenueCorrelationChart** to create charts of different types.

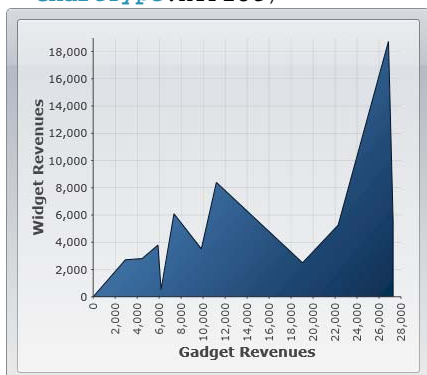
The result should be similar to the images below:



```
BuildRevenueCorrelationChart(
    ChartType.XYPlot)
```



```
BuildRevenueCorrelationChart(
    ChartType.LineSymbols)
```



```
BuildRevenueCorrelationChart(
    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.

## Data-Binding

The previous section described how to create charts by assigning data directly to **DataSeries** objects. This section describes another approach, data binding.

The steps required to create data bound charts are identical to the ones we described above:

1. Choose the chart type (ChartType property).
2. Set up the axes (AxisX and AxisY properties).
3. Add one or more data series (Children collection).
4. 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 (changes are highlighted). 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**, etc). 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 WriteXml method.

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

```
void BuildNorthWindChart(ChartType chartType)
{
    // clear current chart
    _clChart.Reset(true);

    // set chart type
    _clChart.ChartType = chartType;

    // get axes
    Axis valueAxis = _clChart.View.AxisY;
    Axis labelAxis = _clChart.View.AxisX;
    if (chartType == ChartType.Bar)
    {
        valueAxis = _clChart.View.AxisX;
        labelAxis = _clChart.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;

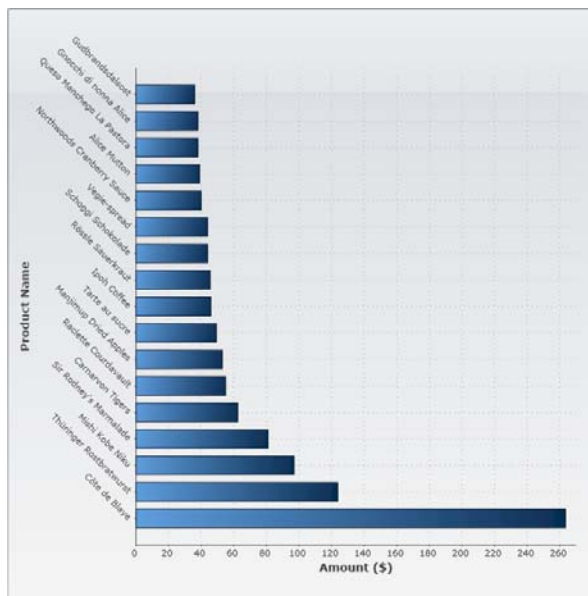
    // show Product Name along x axis
    _c1Chart.Data.ItemNameBinding = new Binding("ProductName");

    // add data series
    var ds = new DataSeries();
    ds.Label = "Unit Price";
    ds.ValueBinding = new Binding("UnitPrice");
    _c1Chart.Data.Children.Add(ds);
}

```

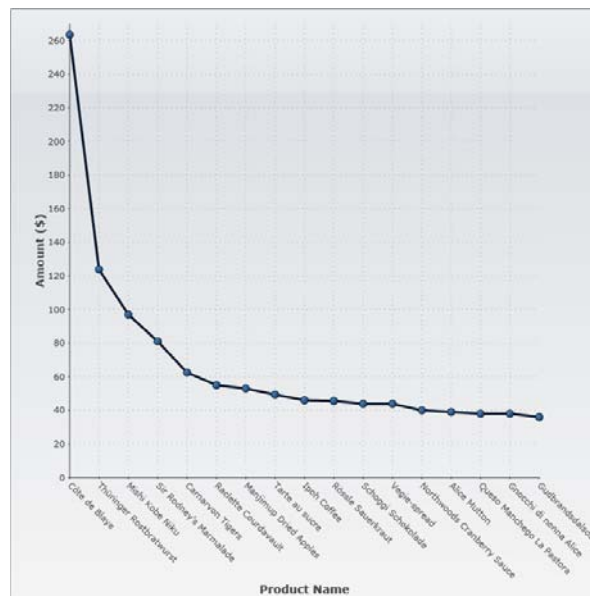
The new code gets the data by retrieving the default view for the products table. It then applies a filter to show only products with prices over \$35 per unit, and sorts the products by unit price.

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";
//_clChart.Data.ItemsSource = dv;
_clChart.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 **Page.xaml** file:

```
<UserControl x:Class="ChartIntro.Page"
    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 Page.xaml.cs file as follows:

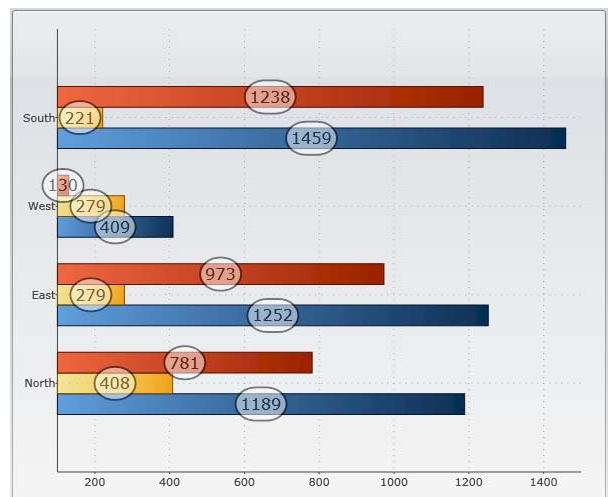
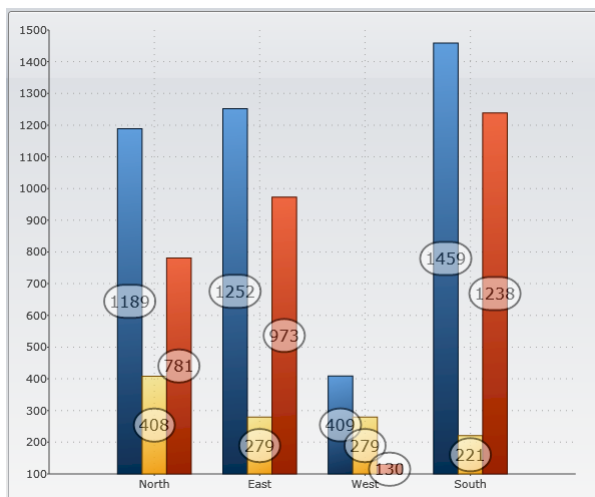
```
// get the data
var data = GetSalesPerRegionData();
_c1Chart.Data.ItemsSource = data;

// 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.Page"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:clchart="clr-
namespace:C1.Silverlight.Chart;assembly=C1.Silverlight.Chart">

  <!-- DataTemplate for chart labels -->
  <UserControl.Resources>
    <clchart:DataPointConverter x:Key="dataPointConverter" />
    <DataTemplate x:Key="chartLabel">
      <Border
        clchart: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>

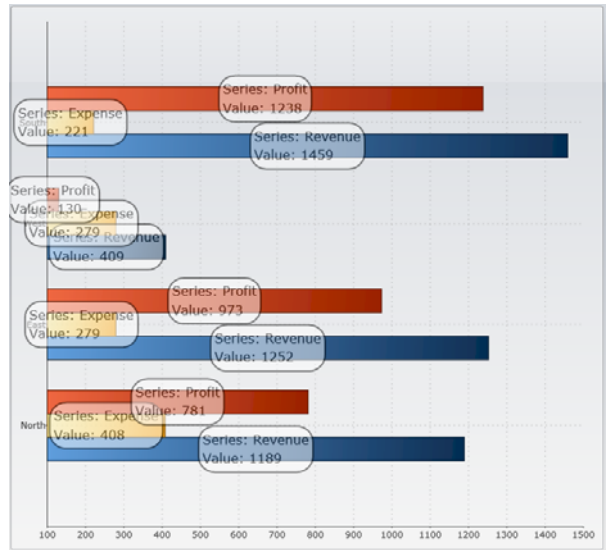
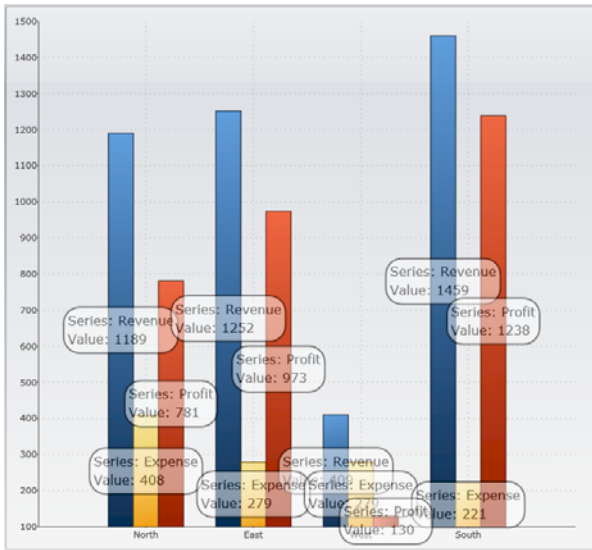
  <!-- main grid -->
  <Grid x:Name="LayoutRoot" Background="White" Margin="10" >
    <!--no changes here -->
  </Grid>
</UserControl
```

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:





# Advanced Topics

## Animation

Each **DataSeries** in a chart is composed of **PlotElement** objects that represent each individual symbol, connector, area, pie slice, etc in the series. The specific type of **PlotElement** depends on the chart type.

You can add animations to your charts by attaching **Storyboard** objects to the plot elements. This is usually done in response to the **DataSeries.Loaded** event, which fires after the **PlotElement** objects have been created and added to the data series.

### 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 _clChart.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, etc.

## 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 events 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 implementation 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 **ClChart** 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 **ClChart**, 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:cl="clr-namespace:C1.Silverlight;assembly=C1.Silverlight"
  xmlns:clchart="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) -->
    <clchart:ClChart x:Name="_clMainChart" />

    <!-- zoom chart (shows entire data range) -->
    <clchart:ClChart x:Name="_clZoomChart" Grid.Row="1" />

    <!-- range slider (selects a range from the zoom chart) -->
    <cl: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(_clMainChart);
  DrawChart(_clZoomChart);
}
```

```

}

// 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.

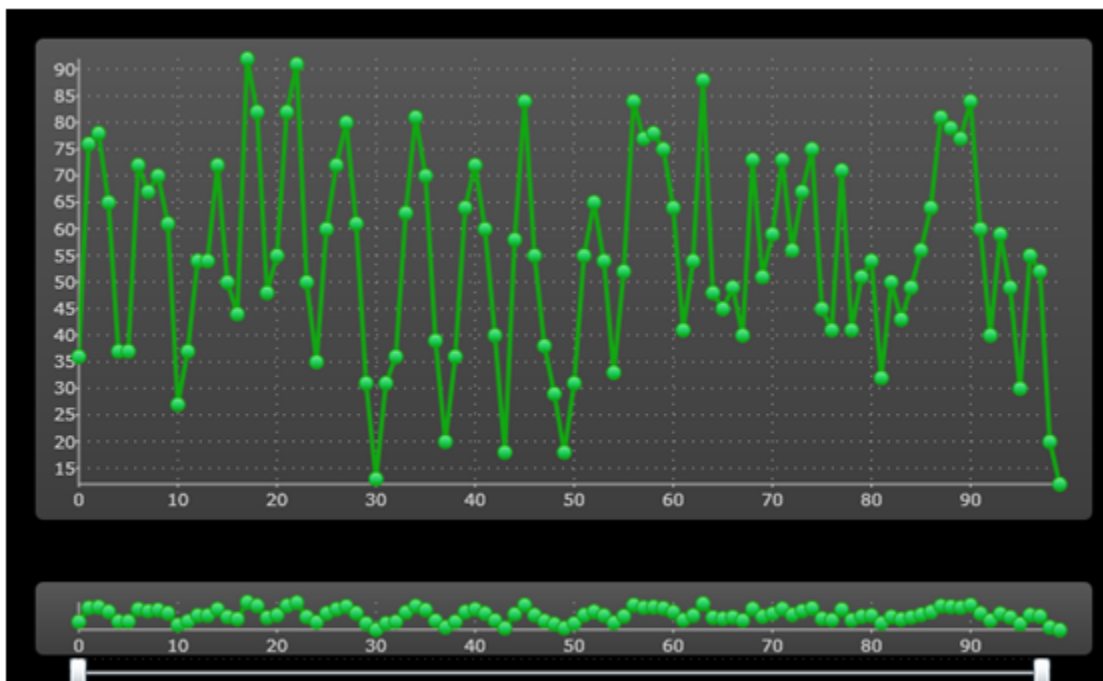
These properties are convenient but not strictly necessary. You could achieve the same results by setting the Max and Min properties on the axis. The advantage of using Scale and Value is that they work over a fixed range between zero and one, which makes the code simpler.

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:

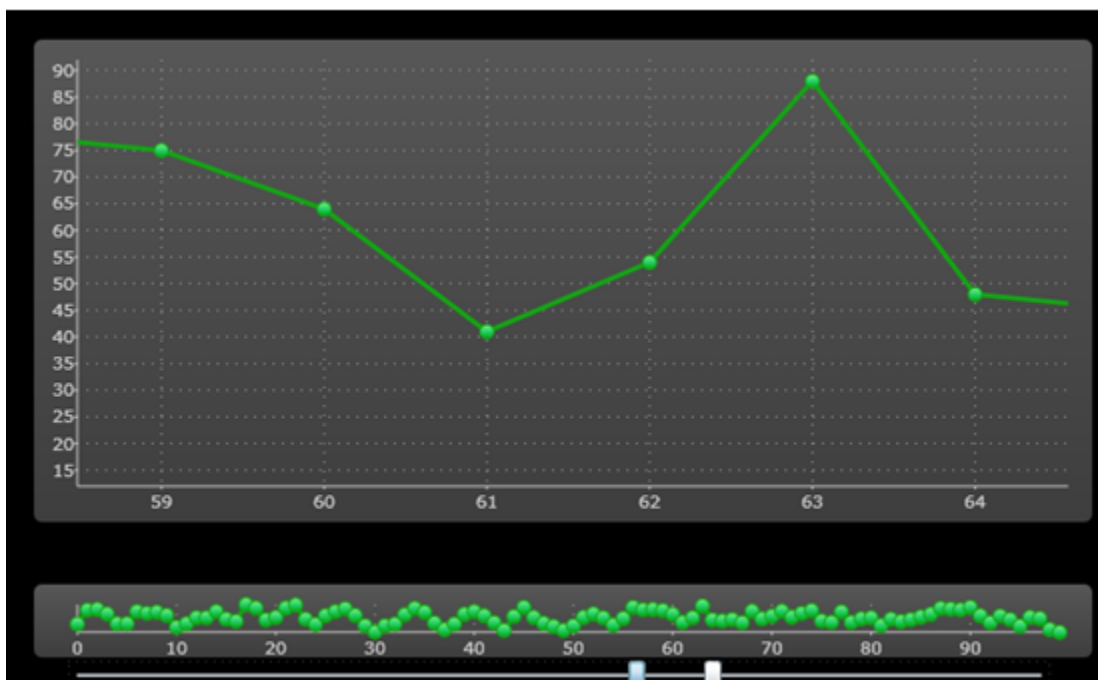
```
// set slider size to match the length of the x axis
protected override Size MeasureOverride(Size availableSize)
{
    Size sz = base.MeasureOverride(availableSize);
    _slider.Width = _clMainChart.View.PlotRect.Width;
    return sz;
}
```

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 _clChart_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 _clChart.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;
            }
        }
    }

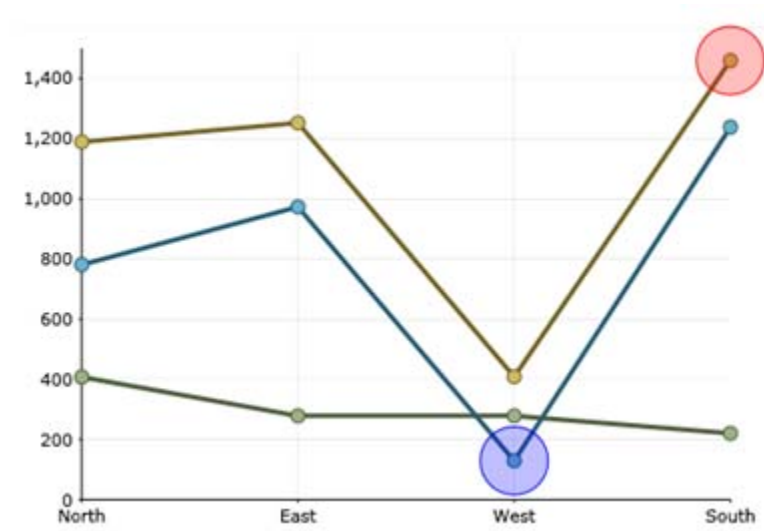
    // position custom element over minimum
    Point ptMin = _clChart.View.PointFromData(new Point(imin, ymin));
    Canvas.SetLeft(_minMark, ptMin.X - _minMark.Width / 2);
    Canvas.SetTop(_minMark, ptMin.Y - _minMark.Height / 2);

    // position custom element over maximum
    Point ptMax = _clChart.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.

---

# Specialized Charts

The chart types discussed so far have been fairly standard. **CIChart** includes many more chart types that can be created following the same steps described above.

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

**CIChart** implements two types of financial chart: **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

**CIChart** 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
    _clChart.Reset(true);

    // set chart type
    _clChart.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
    }
}
```

```
    _clChart.Data.Children.Add(ds);
}

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

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

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

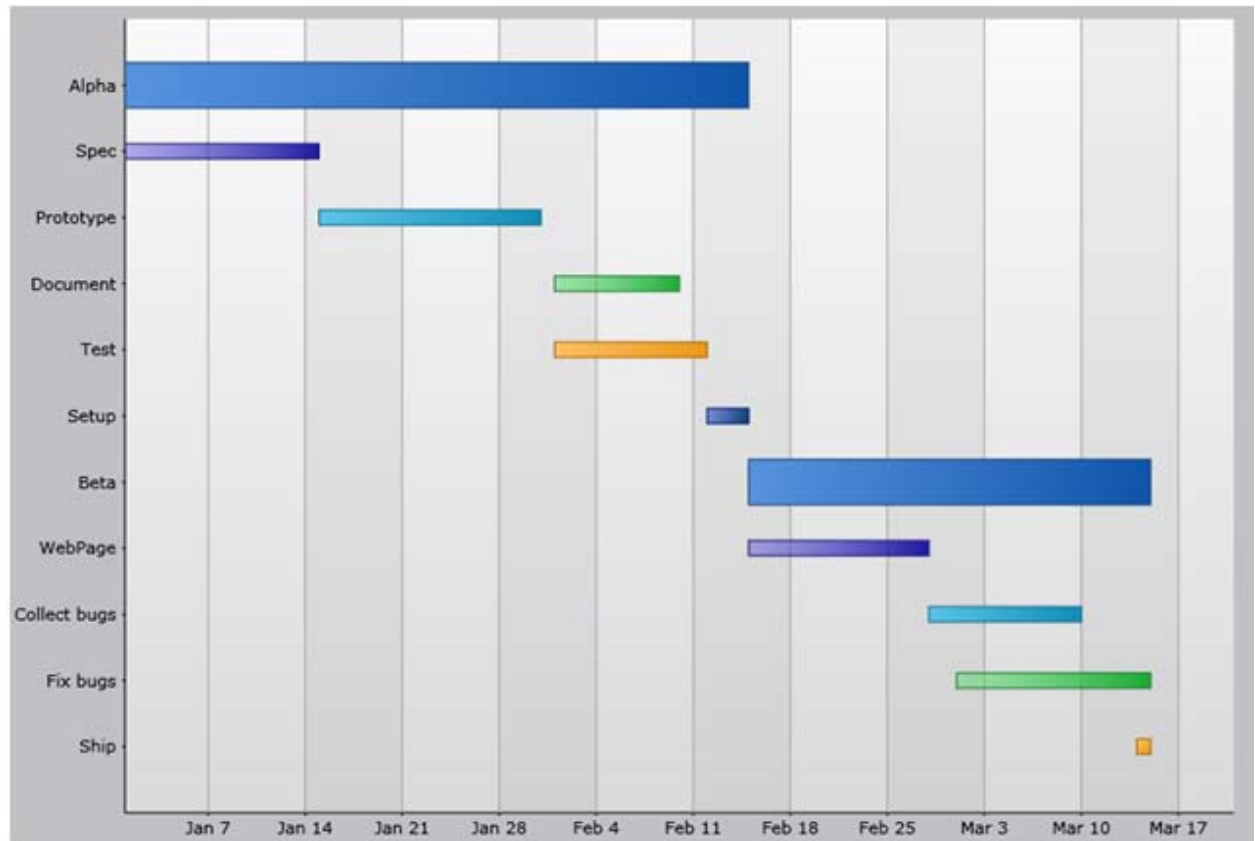
After clearing the **C1Chart** and setting the chart type, the code enumerates the tasks and creates one **HighLowSeries** for each. In addition to setting the series `Label`, `LowValuesSource` and `HighValuesSource` properties, the code uses the `SymbolSize` property to set the height of each bar. In 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:





# 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:clchart="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 -->
    <clchart:C1Chart x:Name="chart0" ChartType="Bar" >

      <!-- populate the chart with two series -->
      <clchart:C1Chart.Data>
        <clchart:ChartData ItemNames="cat1 cat2 cat3 cat4" >
          <clchart:DataSeries Label="s1" Values="1 2 3 4" SymbolFill="Azure"
/>
          <clchart:DataSeries Label="s2" Values="3 2 3 1"
SymbolFill="Crimson" />
        </clchart:ChartData>
      </clchart:C1Chart.Data>

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

    <!-- add a legend -->
    <clchart: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:

