CHAPTER 46

# Introduction to Charts

46

Displaying data in a chart or graph can make data interpretation much easier for users of the applications that you develop with the Adobe Flex product line. Rather than present a simple table of numeric data, you can display a bar, pie, line, or other type of chart using colors, captions, and a two-dimensional representation of your data.

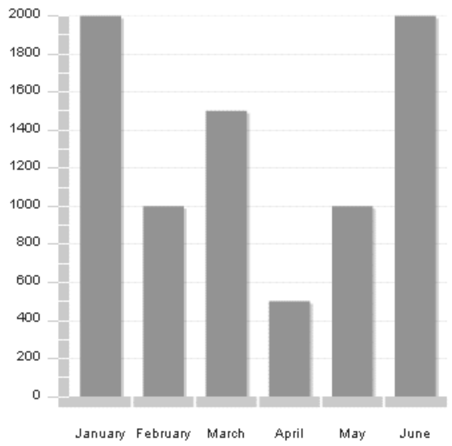
This topic introduces you to the concepts for using the Adobe Flex Charting controls.

## Contents

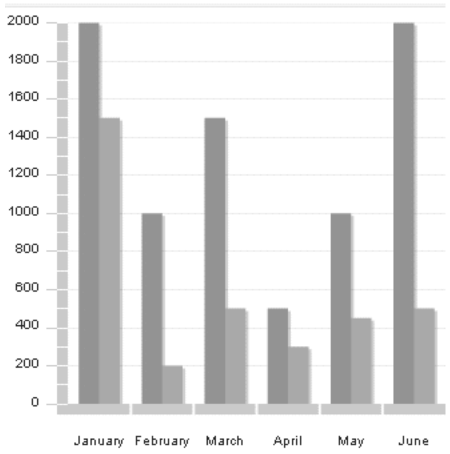
## About charting

Data visualization lets you present data in a way that simplifies data interpretation and data relationships. Charting is one type of data visualization in which you create two-dimensional representations of your data. Flex supports some of the most common types of two-dimensional charts (such as bar, column, and pie charts) and gives you a great deal of control over the appearance of charts.

A simple chart shows a single data series, where a series is a group of related data points. For example, a data series might be monthly sales revenues or daily occupancy rates for a hotel. The following chart shows a single data series that corresponds to sales over several months.



Another chart might add a second data series. For example, you might include the percentage growth of profits over the same four business quarters. The following chart shows two data series—one for sales and one for profit.



For information about installing Flex Charting, see the readme.txt file included with the Flex Charting installer.

# Using the charting controls

Flex Charting lets you create some of the most common chart types, and also lets you customize the appearance of your charts. The charting controls are located in the mx.charts.* package.

The following table lists the supported chart types, the name of the control class, and the name of the series class that you use to define what data appears in each chart.

| Chart type | Chart control class | Chart series class |
|---|---|---|
| Area | AreaChart | AreaSeries |
| Bar | BarChart | BarSeries |
| Bubble | BubbleChart | BubbleSeries |
| Candlestick | CandlestickChart | CandlestickSeries |
| Column | ColumnChart | ColumnSeries |
| HighLowOpenClose | HLOCChart | HLOCSeries |
| Line | LineChart | LineSeries |
| Pie | PieChart | PieSeries |
| Plot | PlotChart | PlotSeries |

All chart controls, except the PieChart class, are subclasses of the CartesianChart class. Cartesian charts are charts that typically represent a set of data points in rectangular-shaped, two-dimensional space. The PieChart class is a subclass of the PolarChart class, which represents data in circular space.

All chart controls inherit basic charting characteristics from the ChartBase class.

A chart control typically has the following structure in MXML:

```
<mx:ChartName>
  <!-- Define one or more series. -->
  <mx:SeriesName/>

  <!-- Define the axes. -->
  <mx:horizontalAxis>
    <mx:AxisType/>
  </mx:horizontalAxis>
  <mx:verticalAxis>
    <mx:AxisType/>
  </mx:verticalAxis>

  <!-- Style the axes and ticks marks. -->
  <mx:horizontalAxisRenderers>
    <mx:AxisRenderer/>
  </mx:horizontalAxisRenderers>
  <mx:verticalAxisRenderers>
    <mx:AxisRenderer/>
  </mx:verticalAxisRenderers/>

  <!-- Add grid lines and other elements to the chart. -->
  <mx:annotationElements>
    <mx:Array/>
  </mx:annotationElements>
  <mx:backgroundElements>
    <mx:Array/>
  </mx:backgroundElements/>
</mx:ChartName>

<!-- Optionally define the legend. -->
<mx:Legend/>
```

The following table describes the parts of the chart in more detail:

| Part | Description |
|---|---|
| Chart | (Required) Defines one or two data providers for the chart. Also defines the chart type and sets data tips, mouse sensitivity, gutter styles, and axis styles. This is the top-level tag for a chart control. All other tags are child tags of this tag. |
| Series | (Required) Defines one or more data series to be displayed on the chart. Also sets the strokes, fills, and renderers (or *skins*) of the data series, as well as the strokes and fills used by the chart's legend for each series.<br>You can also define a second set of series for each chart, to show multiple data series in a single chart.<br>Each series in a chart can have its own data provider. |
| Axes | Sets the axis type (numeric or category). Also defines the axis labels, titles, and style properties such as padding.<br>You can also define axes for the second set of series, if there is one. |
| Axes renderer | (Optional) Sets tick placement and styles, enables or disables labels, and defines axis lines, label rotation, and label gap.<br>You can also define an axis renderer for a second series, if there is one. |
| Elements | (Optional) Defines grid lines and extra elements to appear on the chart. |

For each chart type, Flex supplies a corresponding chart control and chart series. The chart control defines the chart type, the data provider that supplies the chart data, the grid lines, the text for the chart axes, and other properties specific to the chart type. The `dataProvider` property of the chart control determines what data the chart uses.

A *data provider* is a collection of objects. It can be an Array of objects or any object that implements the collections API. A data provider can also be an XMLList object with XML nodes, such as the result of an E4X query.

The chart components use a flat, or list-based, data provider similar to a one-dimensional array. The data provider can contain objects such as Strings and Numbers, or even other objects. For more information on supplying chart data, see "Defining chart data" on page 1612.

You use the chart series to identify which data from the data provider the chart displays. A data provider can contain more data than you want to show in your chart, so you use the chart's series to specify which points you want to use from the data provider. You can specify a single data series or a second series. You can also use the chart series to define the appearance of the data in the chart.

All chart series inherit the data provider from the chart unless they have a data provider explicitly set on themselves. If you set the value of the `dataProvider` property on the chart control, you are not required to set the property value on the series. You can, however, define different data providers for each series in a chart.

For example, to create a pie chart, you use the PieChart control with the PieSeries chart series. To create an area chart, you use the AreaChart control with the AreaSeries chart series, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/BasicAreaOneSeries.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Area Chart">
     <mx:AreaChart id="myChart" dataProvider="{expenses}"
     showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:AreaSeries
                yField="Profit"
                displayName="Profit"
           />
        </mx:series>
     </mx:AreaChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

This example defines an array containing a single `<mx:AreaSeries>` tag. The `<mx:AreaSeries>` tag specifies the single data series that is displayed in the chart.

You can add a second `<mx:AreaSeries>` tag to display two data series, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/BasicArea.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Area Chart">
     <mx:AreaChart id="myChart" dataProvider="{expenses}"
     showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:AreaSeries
                yField="Profit"
                displayName="Profit"
           />
           <mx:AreaSeries
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:AreaChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

You are not required to define a data provider on the chart control. Each series can have its own data provider, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/MultipleDataProviders.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var profit04:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000},
        {Month:"Feb", Profit:1000},
        {Month:"Mar", Profit:1500}
     ]);
     [Bindable]
     public var profit05:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2200},
        {Month:"Feb", Profit:1200},
        {Month:"Mar", Profit:1700}
     ]);
     [Bindable]
     public var profit06:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2400},
        {Month:"Feb", Profit:1400},
        {Month:"Mar", Profit:1900}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart" dataProvider="{profit04}">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="Month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                dataProvider="{profit04}"
                yField="Profit"
                xField="Month"
                displayName="2004"
           />
           <mx:ColumnSeries
                dataProvider="{profit05}"
                yField="Profit"
                xField="Month"
                displayName="2005"
           />
           <mx:ColumnSeries
                dataProvider="{profit06}"
                yField="Profit"
                xField="Month"
                displayName="2006"
```

```
            />
         </mx:series>
      </mx:ColumnChart>
      <mx:Legend dataProvider="{myChart}"/>
   </mx:Panel>
</mx:Application>
```

To dynamically size the chart to the size of the browser window, set the `width` and `height` attributes to a percentage value, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/BasicBarSize.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
   <mx:Script><![CDATA[
      import mx.collections.ArrayCollection;
      [Bindable]
      public var expenses:ArrayCollection = new ArrayCollection([
         {Month:"Jan", Profit:2000, Expenses:1500},
         {Month:"Feb", Profit:1000, Expenses:200},
         {Month:"Mar", Profit:1500, Expenses:500}
      ]);
   ]]></mx:Script>
   <mx:Panel title="Bar Chart" height="1000" width="1000">
      <mx:BarChart id="myChart"
         dataProvider="{expenses}"
         height="100%"
         width="100%"
      >
         <mx:verticalAxis>
            <mx:CategoryAxis
                 dataProvider="{expenses}"
                 categoryField="Month"
            />
         </mx:verticalAxis>
         <mx:series>
            <mx:BarSeries
                 yField="Month"
                 xField="Profit"
                 displayName="Profit"
            />
            <mx:BarSeries
                 yField="Month"
                 xField="Expenses"
                 displayName="Expenses"
            />
         </mx:series>
      </mx:BarChart>
      <mx:Legend dataProvider="{myChart}"/>
   </mx:Panel>
</mx:Application>
```

If you want the chart to resize when the window resizes, set the size of the chart's parent containers using percentage values too.

## About the series classes

The series classes let you specify what data to render in a chart control. All series classes are subclasses of the mx.charts.chartClasses.Series class.

Each chart type has it's own series class; for example, a BarChart control has a BarSeries class that defines the data to render in the BarChart. A PieChart control has a PieSeries.

The primary purpose of a series is to define what data to render in the chart. You use the series to define what field in a data provider the chat should use to render chart items on the X and Y axes. You use the `xField` property (for the horizontal axis) and the `yField` property (for the vertical axis) to define these fields.

Each series is made up of an Array of series items. The classes that define the series items are specific to each series type. For example, a BarSeries is made up of BarSeriesItem objects. A ColumnSeries is made up of ColumnSeriesItem objects. The series items encapsulate all the information about the particular data point, including the minimum value, the fill, the x value and the y value.

When you create a new series, you typically define the `displayName` of that series. This property represents the series to the user in labels such as DataTips.

A BarChart typically specifies one or BarSeries objects that define what set of bars to render in the chart. By default, bars and columns are clustered. However, you can also define alternate ways to group, or "stack", series in the chart control. For example, AreaSeries, ColumnSeries, and BarSeries can be stacked or overlaid. They can also render as 100% charts. You can further control how multiple series are grouped by using sets. For example, for a group of BarSeries objects, you use the BarSet class; for a group of ColumnSeries objects, you use the ColumnSet class. For more information on grouping series, see "Stacking charts" on page 1867.

Most charts use only one kind of series. However, you can specify a second series for a chart, so that a chart control can have a series of bars in addition to a line that "floats" over it. This is useful for rendering trend-lines or showing different types of data on a single chart for comparison analysis. For more information, see "Using multiple data series" on page 1700.

You use the series classes to define the appearance of the chart items. You can change the fill of all series items by using the fill property on the series. In addition, you can define the fill of each item in a series by using the fills property. You can also customize the fill that each chart item has based on its value by using the fillFunction on the series. For more information "Using fills with chart controls" on page 1741.

You can also apply filters to series to give them effects such as drop shadows, blurs, and glows. For more information, see "Using filters with chart controls" on page 1761.

You can add data labels to series items. You do this by setting the value of the `labelPosition` property on the series. For most series, possible values of `labelPosition` are `inside` and `outside`, which draw the labels inside the chart item and outside the chart item, respectively. For a PieSeries, you can also set the `labelPosition` property to other values that include `callout` and `insideWithCallout`.

You can customize data labels by using a `labelFunction`. This callback function takes arguments that define the series item and returns a String that is then rendered on the series item.

For information about adding data labels to your charts, see "Using data labels" on page 1832.

Series also let you set a `minField` value. This property lets you specify a minimum value that the series displays. For more information, see "Using the minField property" on page 1865.

## About the axis classes

Flex Charting supports the following types of axes:

**CategoryAxis** A CategoryAxis class maps a set of values (such as stock ticker symbols, state names, or demographic categories) to the axis. You use the `<mx:CategoryAxis>` tag to define axis labels that are grouped by logical associations and that are not necessarily numeric. For example, the month names used in the chart in "About charting" on page 1593 could be defined as a CategoryAxis class.

**LinearAxis** A LinearAxis class maps numeric data to the axis. You use the `<mx:LinearAxis>` child tag of the `<mx:horizontalAxis>` or `<mx:verticalAxis>` tags to customize the range of values displayed along the axis, and to set the increment between the axis labels of the tick marks.

**LogAxis** A LogAxis class maps numeric data to the axis logarithmically. You use the `<mx:LogAxis>` child tag of the `<mx:horizontalAxis>` or `<mx:verticalAxis>` tags. Labels on the logarithmic axis are even powers of 10.

**DateTimeAxis** A DateTimeAxis class maps time-based values, such as hours, days, weeks, or years, along a chart axis. You use the `<mx:DateTimeAxis>` tag to define the axis labels.

The DateTimeAxis, LogAxis, and LinearAxis are all of type NumericAxis, because they are used to represent numeric values. In many cases, you are required to define only one axis as being a NumericAxis or a CategoryAxis. Flex assumes that all axes not explicitly defined are of type LinearAxis. However, to use decorations such as DataTip labels and legends, you might be required to explicitly define both axes.

For a PlotChart control, both axes are considered a LinearAxis, because the data point is the intersection of two coordinates. So, you are not required to specify either axis, although you can do so to provide additional settings, such as minimum and maximum values.

Each axis can have one or more corresponding AxisRenderer objects (specified by the `horizontalAxisRenderers` or `verticalAxisRenderers` properties) that define the appearance of axis labels and tick marks. In addition to defining formats, you can use an AxisRenderer class to customize the value of the axis labels. For more information, see Chapter 48, "Formatting Charts," on page 1707.

The appearance and contents of *axis labels* are defined by the `<mx:horizontalAxis>` (x-axis) and `<mx:verticalAxis>` (y-axis) tags and the renderers for these tags (`<mx:AxisRenderer>` tags within the `<mx:horizontalAxisRenderers>` and `<mx:verticalAxisRenderers>` tags). These tags not only define the data ranges that appear in the chart, but also map the data points to their names and labels. This latter mapping has a large impact on how the Data Management Service chart renders the values of DataTip labels, axis labels, and tick marks.

By default, Flex uses the chart type and the orientation to calculate the labels that appear along the x-axis and y-axis of the chart. A column chart, for example, has the following default values:

**The x-axis** The minimum value is 0, and the maximum value is the number of items in the data series that is being charted.

**The y-axis** Flex calculates the minimum value on the y-axis to be small enough for the chart data, and the maximum value to be large enough based on the chart data.

For more information about chart axes, see "Working with axes" on page 1795.

## About charting events

The chart controls include events that accommodate user interaction with data points in charts. These events are described in Chapter 50, "Using Events and Effects in Charts," on page 1881.

## Creating charts in ActionScript

You can create, destroy, and manipulate charts using ActionScript just as you can any other Flex component.

When working in Script blocks or in separate ActionScript class files, you must be sure to import all appropriate classes. The following set of import statements defines the most common cases:

```
import mx.collections.*;
import mx.charts.*;
import mx.charts.series.*;
import mx.charts.renderers.*;
import mx.charts.events.*;
```

To create a chart in ActionScript, use the `new` keyword. You can set properties on the chart object as you would in MXML. You assign a data provider with the `dataProvider` property. To add a data series to the chart, you define a new data series of the appropriate type. To apply the series to your chart, use the chart's `series` property. You can specify the category axis settings using the CategoryAxis class. The following example defines a BarChart control with two series:

```
<?xml version="1.0"?>
<!-- charts/CreateChartInActionScript.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="init()">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     import mx.charts.BarChart;
     import mx.charts.series.BarSeries;
     import mx.charts.CategoryAxis;
     import mx.charts.Legend;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);

     public var myChart:BarChart;
     public var series1:BarSeries;
     public var series2:BarSeries;
     public var legend1:Legend;

     public function init():void {
        // Create the chart object and set some
        // basic properties.
        myChart = new BarChart();
        myChart.showDataTips = true;
        myChart.dataProvider = expenses;

        // Define the category axis.
        var vAxis:CategoryAxis = new CategoryAxis();
        vAxis.categoryField = "Month" ;
        vAxis.dataProvider =  expenses;
        myChart.verticalAxis = vAxis;

        // Add the series.
        var mySeries:Array=new Array();
        series1 = new BarSeries();
        series1.xField="Profit";
        series1.yField="Month";
        series1.displayName = "Profit";
        mySeries.push(series1);

        series2 = new BarSeries();
        series2.xField="Expenses";
        series2.yField="Month";
        series2.displayName = "Expenses";
        mySeries.push(series2);

        myChart.series = mySeries;

        // Create a legend.
        legend1 = new Legend();
        legend1.dataProvider = myChart;

        // Attach chart and legend to the display list.
        p1.addChild(myChart);
        p1.addChild(legend1);
     }
  ]]></mx:Script>
  <mx:Panel id="p1" title="Bar Chart Created in ActionScript"/>
</mx:Application>
```

This example replaces the existing Array of series with the new series.

You can use a similar technique to add data series to your charts rather than replacing the existing ones. The following example creates two ColumnSeries and sets their data providers. It then creates an Array that holds the existing chart series, and pushes the new series into that Array. Finally, it sets the value of the chart's `series` property to be the new Array of series.

```
<?xml version="1.0"?>
<!-- charts/AddingSeries.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     import mx.charts.series.ColumnSeries;

     [Bindable]
     public var profit04:ArrayCollection = new ArrayCollection([
        {Month: "Jan", Profit: 2000},
        {Month: "Feb", Profit: 1000},
        {Month: "Mar", Profit: 1500}
     ]);
     [Bindable]
     public var profit05:ArrayCollection = new ArrayCollection([
        {Month: "Jan", Profit: 2200},
        {Month: "Feb", Profit: 1200},
        {Month: "Mar", Profit: 1700}
     ]);
     [Bindable]
     public var profit06:ArrayCollection = new ArrayCollection([
        {Month: "Jan", Profit: 2400},
        {Month: "Feb", Profit: 1400},
        {Month: "Mar", Profit: 1900}
     ]);

     public var series1:ColumnSeries;
     public var series2:ColumnSeries;

     public function addMoreSeries():void {
        if (!series1 || !series2) {
           series1 = new ColumnSeries();
           series1.dataProvider = profit05;
           series1.yField = "Profit";
           series1.xField = "Month";
           series1.displayName = "2005";

           series2 = new ColumnSeries();
           series2.dataProvider = profit06;
           series2.yField = "Profit";
           series2.xField = "Month";
           series2.displayName = "2006";

           var currentSeries:Array = myChart.series;
```

```
                currentSeries.push(series1);
                currentSeries.push(series2);

                myChart.series = currentSeries;
            }
        }
    ]]></mx:Script>
    <mx:Panel title="Column Chart">
        <mx:ColumnChart id="myChart" dataProvider="{profit04}">
            <mx:horizontalAxis>
                <mx:CategoryAxis categoryField="Month"/>
            </mx:horizontalAxis>
            <mx:series>
                <mx:ColumnSeries dataProvider="{profit04}"
                     yField="Profit"
                     xField="Month"
                     displayName="2004"
                />
            </mx:series>
        </mx:ColumnChart>
        <mx:Legend dataProvider="{myChart}"/>
    </mx:Panel>
    <mx:Button id="b1" label="Add More Series To Chart" click=
    "addMoreSeries()"/>
</mx:Application>
```

By using ActionScript, you can also define a variable number of series for your charts. The following example uses E4X syntax to extract an Array of unique names from the data. It then iterates over this Array and builds a new LineSeries for each name.

```
<?xml version="1.0"?>
<!-- charts/VariableSeries.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp();">
    <mx:Script><![CDATA[
            import mx.charts.series.LineSeries;
            import mx.charts.DateTimeAxis;

            [Bindable]
            private var myXML:XML =
                <dataset>
                <item>
                    <who>Tom</who>
                    <when>08/22/2006</when>
                    <hours>5.5</hours>
                </item>
                <item>
                    <who>Tom</who>
                    <when>08/23/2006</when>
                    <hours>6</hours>
                </item>
```

```
        <item>
            <who>Tom</who>
            <when>08/24/2006</when>
            <hours>4.75</hours>
        </item>
        <item>
            <who>Dick</who>
            <when>08/22/2006</when>
            <hours>6</hours>
        </item>
        <item>
            <who>Dick</who>
            <when>08/23/2006</when>
            <hours>8</hours>
        </item>
        <item>
            <who>Dick</who>
            <when>08/24/2006</when>
            <hours>7.25</hours>
        </item>
        <item>
            <who>Jane</who>
            <when>08/22/2006</when>
            <hours>6.5</hours>
        </item>
        <item>
            <who>Jane</who>
            <when>08/23/2006</when>
            <hours>9</hours>
        </item>
        <item>
            <who>Jane</who>
            <when>08/24/2006</when>
            <hours>3.75</hours>
        </item>
        </dataset>;

    public function initApp():void {
        var wholist:Array = new Array();
        for each(var property:XML in myXML.item.who) {
            // Create an Array of unique names.
            if (wholist[property] != property)
                wholist[property] = property;
        }

        // Iterate over names and create a new series
        // for each one.
        for (var s:String in wholist) {
            // Use all items whose name matches s.
            var localXML:XMLList = myXML.item.(who==s);
```

```
                // Create the new series and set its properties.
                var localSeries:LineSeries = new LineSeries();
                localSeries.dataProvider = localXML;
                localSeries.yField = "hours";
                localSeries.xField = "when";

                // Set values that show up in dataTips and Legend.
                localSeries.displayName = s;

                // Back up the current series on the chart.
                var currentSeries:Array = myChart.series;
                // Add the new series to the current Array of series.
                currentSeries.push(localSeries);
                // Add the new Array of series to the chart.
                myChart.series = currentSeries;
            }

            // Create a DateTimeAxis horizontal axis.
            var hAxis:DateTimeAxis = new DateTimeAxis();
            hAxis.dataUnits = "days";
            // Set this to false to display the leftmost label.
            hAxis.alignLabelsToUnits = false;
            // Take the date in its current format and create a Date
            // object from it.
            hAxis.parseFunction = createDate;
            myChart.horizontalAxis = hAxis;
        }

        public function createDate(s:String):Date {
            // Reformat the date input to create Date objects
            // for the axis.
            var a:Array = s.split("/");

            // The existing String s is in the format "MM/DD/YYYY".
            // To create a Date object, you pass "YYYY,MM,DD",
            // where MM is zero-based, to the Date() constructor.
            var newDate:Date = new Date(a[2],a[0]-1,a[1]);
            return newDate;
        }
    ]]></mx:Script>

  <mx:Panel title="Line Chart with Variable Number of Series">
     <mx:LineChart id="myChart" showDataTips="true"/>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

## Defining chart data

The chart controls have a `dataProvider` property that defines the data for the chart. The data provider creates a level of abstraction between Flex components and the data that you use to populate them. You can populate multiple charts from the same data provider, switch data providers for a chart at run time, and modify the data provider so that changes are reflected by all charts using the data provider.

### Using chart data

To use the data from a data provider in your chart control, you map the `xField` and `yField` properties of the chart series to the fields in the data provider. The `xField` property defines the data for the horizontal axis, and the `yField` property defines the data for the vertical axis.

For example, assume your data provider has the following structure:

```
{Month: "Feb", Profit: 1000, Expenses: 200, Amount: 60}
```

You can use the `Profit` and `Expenses` fields and ignore the `Month` field by mapping the `xField` property of the series object to one field and the `yField` property of the series object to another field, as the following example shows:

```
<mx:PlotSeries xField="Profit" yField="Expenses"/>
```

The result is that each data point is the intersection of the `Profit` and `Expenses` fields from the data provider.

To place the data points into a meaningful grouping, you can choose a separate property of the data provider as the `categoryField`. In this case, to sort each data point by month, you map the `Month` field to the `categoryField` property of the horizontal axis:

```
<mx:horizontalAxis>
  <mx:CategoryAxis dataProvider="{expenses}" categoryField="Month"/>
</mx:horizontalAxis>
```

In some cases, depending on the type of chart and the type of data you are representing, you use either the `xField` property or the `yField` property to define the data series. In a ColumnChart control, for example, the `yField` property defines the height of the column. You do not have to specify an `xField` property. To get an axis label for each column, you specify a `categoryField` property for the `horizontalAxis`.

When you use chart data, keep the following in mind:

- You usually match a series with a data provider field if you want to display that series. However, this is not always true. If you do not specify an `xField` for a ColumnSeries, Flex assumes the index is the value. If you do not specify a `yField`, Flex assumes the data provider is a collection of *y* values, rather than a collection of objects that have *y* values. For example, the following series renders correctly for a ColumnChart control:
  ```
  <mx:ColumnSeries dataProvider="{[1,2,3,4,5]}"/>
  ```
- Some series use only one field from the data provider, while others can use two or more. For example, you specify only a `field` property for a PieSeries object, but you can specify an `xField` and a `yField` for a PlotSeries object and an `xField`, `yField`, and `radiusField` for a BubbleSeries object.
- Most of the series can determine suitable defaults for their nonprimary dimensions if no field is specified. For example, if you do not explicitly set an `xField` for the ColumnSeries, LineSeries, and AreaSeries, Flex maps the data to the chart's categories in the order in which the data appears in the data provider. Similarly, a BarSeries maps the data to the categories if you do not set a `yField`.

For a complete list of the fields that each data series can take, see the data series' entry in *Adobe Flex Language Reference*. For more information on data providers, see "Using data provider controls" on page 270.

## Types of chart data

You can supply data to a data provider in the following ways:

- Define it in a `<mx:Script>` block.
- Define it in an external XML, ActionScript, or text file.
- Return it by using a WebService call.
- Return it by using a RemoteObject component.
- Return it by using an HTTPService component.
- Define it in MXML.

For more information on data providers, see Chapter 8, "Using Data Providers and Collections," on page 171.

## Using static Arrays as data providers

Using a static Array of objects for the data provider is the simplest approach. You typically create an Array of objects, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/ArrayOfObjectsDataProvider.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     private var expenses:Array = [
        {Month:"January",Profit:2000,Expenses:1500,Amount:450},
        {Month:"February",Profit:1000,Expenses:200,Amount:600},
        {Month:"March",Profit:1500,Expenses:500,Amount:300},
        {Month:"April",Profit:500,Expenses:300,Amount:500},
        {Month:"May",Profit:1000,Expenses:450,Amount:250},
        {Month:"June",Profit:2000,Expenses:500,Amount:700}
     ];

  ]]></mx:Script>
  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

You can also use MXML to define the content of an Array, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/ArrayOfMXMLObjectsDataProvider.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Array id="expenses">
     <mx:Object
        Month="January"
        Profit="2000"
        Expenses="1500"
        Amount="450"
     />
     <mx:Object
        Month="February"
        Profit="1000"
        Expenses="200"
        Amount="600"
     />
     <mx:Object
        Month="March"
        Profit="1500"
        Expenses="500"
        Amount="300"
     />
     <mx:Object
        Month="April"
        Profit="500"
        Expenses="300"
        Amount="500"
     />
     <mx:Object
        Month="May"
        Profit="1000"
        Expenses="450"
        Amount="250"
     />
     <mx:Object
        Month="June"
        Profit="2000"
        Expenses="500"
        Amount="700"
     />
  </mx:Array>

  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

You can also define objects in MXML with a more verbose syntax, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/ArrayOfVerboseMXMLObjects.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Array id="expenses">
     <mx:Object>
       <mx:Month>January</mx:Month>
       <mx:Profit>2000</mx:Profit>
       <mx:Expenses>1500</mx:Expenses>
       <mx:Amount>450</mx:Amount>
     </mx:Object>
     <mx:Object>
       <mx:Month>February</mx:Month>
       <mx:Profit>1000</mx:Profit>
       <mx:Expenses>200</mx:Expenses>
       <mx:Amount>600</mx:Amount>
     </mx:Object>
     <mx:Object>
       <mx:Month>March</mx:Month>
       <mx:Profit>1500</mx:Profit>
       <mx:Expenses>500</mx:Expenses>
       <mx:Amount>300</mx:Amount>
     </mx:Object>
     <mx:Object>
       <mx:Month>April</mx:Month>
       <mx:Profit>500</mx:Profit>
       <mx:Expenses>300</mx:Expenses>
       <mx:Amount>300</mx:Amount>
     </mx:Object>
     <mx:Object>
       <mx:Month>May</mx:Month>
       <mx:Profit>1000</mx:Profit>
       <mx:Expenses>450</mx:Expenses>
       <mx:Amount>250</mx:Amount>
     </mx:Object>
     <mx:Object>
       <mx:Month>June</mx:Month>
       <mx:Profit>2000</mx:Profit>
       <mx:Expenses>500</mx:Expenses>
       <mx:Amount>700</mx:Amount>
     </mx:Object>
  </mx:Array>

  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
            />
        </mx:horizontalAxis>
        <mx:series>
            <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
            />
            <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
            />
        </mx:series>
    </mx:ColumnChart>
    <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

A disadvantage of using a simple Array as a chart's data provider is that you can use only the methods of the Array class to manipulate the data. In addition, when you use an Array as a data provider, the data in it must be static. Even if you make the Array bindable, when data in an Array changes, the chart does not reflect those changes. For more robust data manipulation and data binding, you can use a collection for the chart data provider, as described in "Using collections as data providers" on page 1618.

## Using collections as data providers

Collections are a more robust data provider mechanism than Arrays. They provide operations that include the insertion and deletion of objects as well as sorting and filtering. Collections also support change notification. An ArrayCollection object provides an easy way to expose an Array as an ICollectionView or IList interface.

As with Arrays, you can use MXML to define the contents of a collection, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/ArrayCollectionOfMXMLObjectsDataProvider.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:ArrayCollection id="expenses">
     <mx:Object
        Month="January"
        Profit="2000"
        Expenses="1500"
        Amount="450"
     />
     <mx:Object
        Month="February"
        Profit="1000"
        Expenses="200"
        Amount="600"
     />
     <mx:Object
        Month="March"
        Profit="1500"
        Expenses="500"
        Amount="300"
     />
     <mx:Object
        Month="April"
        Profit="500"
        Expenses="300"
        Amount="500"
     />
     <mx:Object
        Month="May"
        Profit="1000"
        Expenses="450"
        Amount="250"
     />
     <mx:Object
        Month="June"
        Profit="2000"
        Expenses="500"
        Amount="700"
     />
  </mx:ArrayCollection>

  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
            />
        </mx:horizontalAxis>
        <mx:series>
            <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
            />
            <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
            />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

Or you can define a complex object in MXML:

```
<?xml version="1.0"?>
<!-- charts/ArrayOfVerboseMXMLObjects.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:ArrayCollection id="expenses">
     <mx:Object>
       <mx:Month>January</mx:Month>
       <mx:Profit>2000</mx:Profit>
       <mx:Expenses>1500</mx:Expenses>
       <mx:Amount>450</mx:Amount>
     </mx:Object>
     <mx:Object>
       <mx:Month>February</mx:Month>
       <mx:Profit>1000</mx:Profit>
       <mx:Expenses>200</mx:Expenses>
       <mx:Amount>600</mx:Amount>
     </mx:Object>
     <mx:Object>
       <mx:Month>March</mx:Month>
       <mx:Profit>1500</mx:Profit>
       <mx:Expenses>500</mx:Expenses>
       <mx:Amount>300</mx:Amount>
     </mx:Object>
     <mx:Object>
       <mx:Month>April</mx:Month>
       <mx:Profit>500</mx:Profit>
       <mx:Expenses>300</mx:Expenses>
       <mx:Amount>300</mx:Amount>
     </mx:Object>
     <mx:Object>
       <mx:Month>May</mx:Month>
       <mx:Profit>1000</mx:Profit>
       <mx:Expenses>450</mx:Expenses>
       <mx:Amount>250</mx:Amount>
     </mx:Object>
     <mx:Object>
       <mx:Month>June</mx:Month>
       <mx:Profit>2000</mx:Profit>
       <mx:Expenses>500</mx:Expenses>
       <mx:Amount>700</mx:Amount>
     </mx:Object>
  </mx:ArrayCollection>

  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

You can create an ArrayCollection object in ActionScript. If you define an ArrayCollection in this way, ensure that you import the mx.collections.ArrayCollection class, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/ArrayCollectionOfObjects.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     private var expenses:ArrayCollection = new ArrayCollection([
        {Month:"January", Profit:2000, Expenses:1500, Amount:450},
        {Month:"February", Profit:1000, Expenses:200, Amount:600},
        {Month:"March", Profit:1500, Expenses:500, Amount:300},
        {Month:"April", Profit:500, Expenses:300, Amount:500},
        {Month:"May", Profit:1000, Expenses:450, Amount:250},
        {Month:"June", Profit:2000, Expenses:500, Amount:700}
     ]);

  ]]></mx:Script>
  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

If your data is in an Array, you can pass the Array to the ArrayCollection's constructor to convert it to an ArrayCollection. The following example creates an Array, and then converts it to an ArrayCollection:

```
<?xml version="1.0"?>
<!-- charts/ArrayConvertedToArrayCollection.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     private var expenses:Array = [
        {Month:"January", Profit:2000, Expenses:1500, Amount:450},
        {Month:"February", Profit:1000, Expenses:200, Amount:600},
        {Month:"March", Profit:1500, Expenses:500, Amount:300},
        {Month:"April", Profit:500, Expenses:300, Amount:500},
        {Month:"May", Profit:1000, Expenses:450, Amount:250},
        {Month:"June", Profit:2000, Expenses:500, Amount:700}
     ];

     [Bindable]
     public var expensesAC:ArrayCollection =
        new ArrayCollection(expenses);

  ]]></mx:Script>

  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart" dataProvider="{expensesAC}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expensesAC}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

Similarly, you can use an `<mx:ArrayCollection>` tag to perform the conversion:

```
<?xml version="1.0"?>
<!-- charts/ArrayConvertedToArrayCollectionMXML.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     private var expenses:Array = [
        {Month:"January", Profit:2000, Expenses:1500, Amount:450},
        {Month:"February", Profit:1000, Expenses:200, Amount:600},
        {Month:"March", Profit:1500, Expenses:500, Amount:300},
        {Month:"April", Profit:500, Expenses:300, Amount:500},
        {Month:"May", Profit:1000, Expenses:450, Amount:250},
        {Month:"June", Profit:2000, Expenses:500, Amount:700}
     ];
  ]]></mx:Script>

  <mx:ArrayCollection id="expensesAC" source="{expenses}"/>

  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart" dataProvider="{expensesAC}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expensesAC}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

The data in ArrayCollections can be bound to the chart's data provider so that the data can be updated in real-time. The following example creates an object with elapsed time and total memory usage every second. It then pushes that new object onto an ArrayCollection that is used as the data provider for a line chart. As a result, the chart itself updates every second showing memory usage of Flash Player over time.

```
<?xml version="1.0"?>
<!-- charts/RealTimeArrayCollection.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" initialize=
"initTimer()">
  <mx:Script><![CDATA[
  import flash.utils.Timer;
  import flash.events.TimerEvent;
  import mx.collections.ArrayCollection;

  [Bindable]
  public var memoryUsage:ArrayCollection = new ArrayCollection();

  public function initTimer():void {
     // The first parameter in the Timer constructor
     // is the interval, in milliseconds.
     // The second parameter is how many times to run (0 is
     // infinity).
     var myTimer:Timer = new Timer(1000, 0);

     // Add the listener for the timer event.
     myTimer.addEventListener("timer", timerHandler);
     myTimer.start();
  }

  public function timerHandler(event:TimerEvent):void {
     var o:Object = new Object();

     // Get the number of milliseconds since Flash Player started.
     o.time = getTimer();

     // Get the total memory Flash Player is using.
     o.memory = flash.system.System.totalMemory;
     trace(o.time + ":" + o.memory);

     // Add new object to the ArrayCollection, which is bound
     // to the chart's data provider.
     memoryUsage.addItem(o);
  }
  ]]></mx:Script>

  <mx:LineChart id="chart" dataProvider="{memoryUsage}"
     showDataTips="true">
     <mx:horizontalAxis>
```

```
            <mx:LinearAxis/>
         </mx:horizontalAxis>
         <mx:verticalAxis>
            <mx:LinearAxis minimum="5000000"/>
         </mx:verticalAxis>
         <mx:series>
            <mx:LineSeries yField="memory"/>
         </mx:series>
   </mx:LineChart>
</mx:Application>
```

Data collections can be paged, which means that data is sent to the client in chunks as the application requests it. But Flex Charting components display all of the data all of the time, by default. As a result, when you use data collections with charts, you should disable the paging features or use non-paged views of the data collection for chart data. For more information on using collections, see "About collections" on page 177.

## Using an XML file as a data provider

You can define data provider data in a structured file. The following example shows the contents of the data.xml file:

```
<data>
<result month="Jan-04">
<apple>81768</apple>
<orange>60310</orange>
<banana>43357</banana>
</result>
<result month="Feb-04">
<apple>81156</apple>
<orange>58883</orange>
<banana>49280</banana>
</result>
</data>
```

You can load the file directly as a source of a Model, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/XMLFileDataProvider.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Model id="results" source="../assets/data.xml"/>
  <mx:Panel title="Line Chart">
     <mx:LineChart id="chart" dataProvider="{results.result}">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries yField="banana" displayName="Banana"/>
           <mx:LineSeries yField="apple" displayName="Apple"/>
           <mx:LineSeries yField="orange" displayName="Orange"/>
        </mx:series>
     </mx:LineChart>
  </mx:Panel>
</mx:Application>
```

To use an ArrayCollection as the chart's data provider, you convert the Model to one, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/XMLFileToArrayCollectionDataProvider.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="100%" height="100%">
  <mx:Script>
     import mx.utils.ArrayUtil;
  </mx:Script>

  <mx:Model id="results" source="../assets/data.xml"/>
  <mx:ArrayCollection id="myAC"
        source="{ArrayUtil.toArray(results.result)}"
  />

  <mx:Panel title="Line Chart">
     <mx:LineChart id="chart" dataProvider="{myAC}">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries yField="banana" displayName="Banana"/>
           <mx:LineSeries yField="apple" displayName="Apple"/>
           <mx:LineSeries yField="orange" displayName="Orange"/>
        </mx:series>
     </mx:LineChart>
  </mx:Panel>
</mx:Application>
```

You can also define the XML file as a URL for an HTTPService component, and then bind the HTTPService result directly to the chart's data provider, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/HTTPServiceDataProvider.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    width="100%" height="100%" creationComplete="srv.send()">
  <mx:HTTPService id="srv" url="../assets/data.xml"/>

  <mx:Panel title="Line Chart">
     <mx:LineChart id="chart"
        dataProvider="{srv.lastResult.data.result}"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries yField="apple" name="Apple"/>
           <mx:LineSeries yField="orange" name="Orange"/>
           <mx:LineSeries yField="banana" name="Banana"/>
        </mx:series>
     </mx:LineChart>
  </mx:Panel>
</mx:Application>
```

To use an ArrayCollection, you convert the HTTPService result to an ArrayCollection, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/HTTPServiceToArrayCollectionDataProvider.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="srv.send()">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var myData:ArrayCollection;
  ]]></mx:Script>

  <mx:HTTPService
     id="srv"
     url="../assets/data.xml"
     useProxy="false"
     result="myData=ArrayCollection(srv.lastResult.data.result)"
  />
  <mx:Panel title="Line Chart">
     <mx:LineChart id="chart" dataProvider="{myData}">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries yField="apple" name="Apple"/>
           <mx:LineSeries yField="orange" name="Orange"/>
           <mx:LineSeries yField="banana" name="Banana"/>
        </mx:series>
     </mx:LineChart>
  </mx:Panel>
</mx:Application>
```

You can also set the result format of the HTTPService to E4X, and then use it as a source for an XMLListCollection object, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/HTTPServiceToXMLListCollection.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="srv.send()">
  <mx:Script><![CDATA[
     import mx.utils.ArrayUtil;
  ]]></mx:Script>

  <mx:HTTPService id="srv"
    url="../assets/data.xml"
    resultFormat="e4x"
  />

  <mx:XMLListCollection id="myAC"
    source="{srv.lastResult.result}"
  />

  <mx:Panel title="Line Chart">
     <mx:LineChart id="chart" dataProvider="{myAC}">
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries yField="apple" name="Apple"/>
           <mx:LineSeries yField="orange" name="Orange"/>
           <mx:LineSeries yField="banana" name="Banana"/>
        </mx:series>
     </mx:LineChart>
  </mx:Panel>
</mx:Application>
```

## Randomly generating chart data

A useful way to create data for use in sample charts is to generate random data. The following example generates test data for use with the chart controls:

```
<?xml version="1.0"?>
<!-- charts/RandomDataGeneration.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
  <mx:Script><![CDATA[
     import mx.collections.*;

     // Define data provider array for the chart data.
     [Bindable]
     public var dataSet:ArrayCollection;

     // Define the number of elements in the array.
     public var dsLength:Number = 10;

     public function initApp():void {
        // Initialize data provider array.
        dataSet = new ArrayCollection(genData());
     }

     public function genData():Array {
        var result:Array = [];

        for (var i:int=0;i<dsLength;i++) {
           var localVals:Object = {
                valueA:Math.random()*100,
                valueB:Math.random()*100,
                valueX:Math.random()*100,
                valueY:Math.random()*100
           };

           // Push new object onto the data array.
           result.push(localVals);
        }
        return result;
     }
  ]]></mx:Script>

  <mx:Panel title="Plot Chart">
     <mx:PlotChart id="myChart" dataProvider="{dataSet}">
        <mx:series>
           <mx:PlotSeries
                xField="valueX"
                yField="valueY"
                displayName="Series 1"
           />
```

```
          <mx:PlotSeries
               xField="valueA"
               yField="valueB"
               displayName="Series 2"
          />
       </mx:series>
     </mx:PlotChart>
     <mx:Legend id="l1" dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

## Changing chart data at run time

Using ActionScript, you can change a charting control's data at run time by using a variety of methods.

You can change a chart or a series data provider. The following example binds the data provider to a local variable. It then toggles the chart's data provider using that local variable when the user clicks the button.

```
<?xml version="1.0"?>
<!-- charts/ChangeDataProvider.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);

     [Bindable]
     public var expenses2:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2400, Expenses:1509, Amount:950},
        {Month:"Feb", Profit:3000, Expenses:2200, Amount:400},
        {Month:"Mar", Profit:3500, Expenses:1200, Amount:200}
     ]);

     [Bindable]
     public var dp:ArrayCollection = expenses;

     public function changeDataProvider():void {
        if (dp==expenses) {
           dp = expenses2;
        } else {
           dp = expenses;
        }
     }
  ]]></mx:Script>
  <mx:Panel title="Line Chart">
     <mx:LineChart id="myChart" dataProvider="{dp}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{dp}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries
                yField="Profit"
                displayName="Profit"
           />
           <mx:LineSeries
                yField="Expenses"
```

```
                    displayName="Expenses"
               />
               <mx:LineSeries
                    yField="Amount"
                    displayName="Amount"
               />
          </mx:series>
     </mx:LineChart>
     <mx:Legend dataProvider="{myChart}"/>
     <mx:Button label="Change Data Provider"
          click="changeDataProvider()"
     />
  </mx:Panel>
</mx:Application>
```

You can add or remove a data point from a series. The following example adds an item to the existing data provider when the user clicks the button:

```
<?xml version="1.0"?>
<!-- charts/AddDataItem.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns="*">
    <mx:Script><![CDATA[
        import mx.collections.ArrayCollection;
        [Bindable]
        public var dpac:ArrayCollection =  new ArrayCollection([
           {A:2000},
           {A:3000},
           {A:4000},
           {A:4000},
           {A:3000},
           {A:2000},
           {A:6000}
        ]);

        public function addDataItem():void {
            var o:Object = {"A":2000};
            dpac.addItem(o);
        }
    ]]></mx:Script>
  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart"
        height="400"
        width="800"
        dataProvider="{dpac}"
     >
        <mx:series>
            <mx:ColumnSeries yField="A" displayName="Series 1"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
     <mx:Button label="Add Data Item" click="addDataItem();"/>
  </mx:Panel>
</mx:Application>
```

You can also change the fields of a series to change chart data at run time. You do this by changing the value of the data provider field (such as `xField` or `yField`) on the series object. To get a reference to the series, you use the series's `id` property or the chart control's series index. The following example toggles the data series when the user clicks on the Change Series button:

```
<?xml version="1.0"?>
<!-- charts/ToggleSeries.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" initialize=
"initApp();">
  <mx:Script><![CDATA[
     [Bindable]
     public var dataSet:Array;
     public var myStates:Array =
        ["Wisconsin","Ohio","Arizona","Penn"];
     public var curSeries:String;
     public function initApp():void {
        var newData:Array = [];
        for(var i:int=0;i<myStates.length;i++) {
           newData[i] = {
            Apples: Math.floor(Math.random()*150),
            Oranges: Math.floor(Math.random()*150),
            myState: myStates[i]
           }
        }
        dataSet = newData;
        curSeries = "apples";
     }
     public function changeData():void {
        var series:Object = myChart.series[0];
        if (curSeries == "apples") {
           curSeries="oranges";
           series.yField = "Oranges";
           series.displayName = "Oranges";
           series.setStyle("fill",0xFF9933);
        } else {
           curSeries="apples";
           series.yField = "Apples";
           series.displayName = "Apples";
           series.setStyle("fill",0xFF0000);
        }
     }
  ]]></mx:Script>
  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart"
        dataProvider="{dataSet}"
        showDataTips="true"
     >
        <mx:horizontalAxis>
```

```
           <mx:CategoryAxis
                dataProvider="{dataSet}"
                categoryField="myState"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                yField="Apples"
                displayName="Apples"
            >
            <mx:fill>
                <mx:SolidColor color="0xFF0000"/>
            </mx:fill>
           </mx:ColumnSeries>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
     <mx:Button id="b1"
        label="Change Series"
        click="changeData()"
     />
  </mx:Panel>
</mx:Application>
```

You can take advantage of data binding to make your chart reflect data changes in real time. The following example uses a Timer to define the intervals at which it checks for new data. Because the chart's data provider is bound to an ArrayCollection, whenever a new data point is added to the collection, the chart is automatically updated.

```
<?xml version="1.0"?>
<!-- charts/WatchingCollections.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    initialize="initData();">
    <mx:Script><![CDATA[
        import mx.collections.ArrayCollection;

        [Bindable]
        public var dataSet:ArrayCollection;

        [Bindable]
        public var revenue:Number = 100;

        private var t:Timer;

        private function initData():void {
            dataSet = new ArrayCollection();
            t = new Timer(500);
            t.addEventListener(TimerEvent.TIMER, addData);
            t.start();
        }

        private function addData(e:Event):void {
            dataSet.addItem( { revenue: revenue } );
            revenue += Math.random() * 10 - 5;
        }
    ]]></mx:Script>

    <mx:SeriesInterpolate id="interp"
        elementOffset="0"
        duration="300"
        minimumElementDuration="0"
    />

    <mx:Panel title="Line Chart">
        <mx:LineChart id="myChart" dataProvider="{dataSet}">
            <mx:series>
                <mx:LineSeries
                    yField="revenue"
                    showDataEffect="{interp}"
                />
            </mx:series>
            <mx:horizontalAxis>
                <mx:LinearAxis autoAdjust="false"/>
            </mx:horizontalAxis>
        </mx:LineChart>
    </mx:Panel>
</mx:Application>
```

You can also use system values to update charts at run time. The following example tracks the value of the `totalMemory` property in a LineChart control in real time:

```
<?xml version="1.0"?>
<!-- charts/MemoryGraph.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" initialize=
"initTimer()">
  <mx:Script><![CDATA[
  import flash.utils.Timer;
  import flash.events.TimerEvent;
  import mx.collections.ArrayCollection;

  [Bindable]
  public var memoryUsage:ArrayCollection = new ArrayCollection();

        public function initTimer():void {
           // The first parameter in the Timer constructor
           // is the interval, in milliseconds. The second
           // parameter is how many times to run (0 is
           // infinity).
           var myTimer:Timer = new Timer(1000, 0);

           // Add the listener for the timer event.
           myTimer.addEventListener("timer", timerHandler);
           myTimer.start();
        }

        public function timerHandler(event:TimerEvent):void {
           var o:Object = new Object();

           // Get the number of milliseconds since Flash
           // Player started.
           o.time = getTimer();

           // Get the total memory Flash Player is using.
           o.memory = flash.system.System.totalMemory;

           // Add new object to the ArrayCollection, which
           // is bound to the chart's data provider.
           memoryUsage.addItem(o);
        }
  ]]></mx:Script>

  <mx:LineChart id="chart"
        dataProvider="{memoryUsage}"
        showDataTips="true"
  >
     <mx:horizontalAxis>
        <mx:LinearAxis/>
     </mx:horizontalAxis>
```

```
      <mx:verticalAxis>
         <mx:LinearAxis minimum="5000000"/>
      </mx:verticalAxis>
      <mx:series>
         <mx:Array>
            <mx:LineSeries yField="memory"/>
         </mx:Array>
      </mx:series>
  </mx:LineChart>
</mx:Application>
```

CHAPTER 47

# Chart Types

47

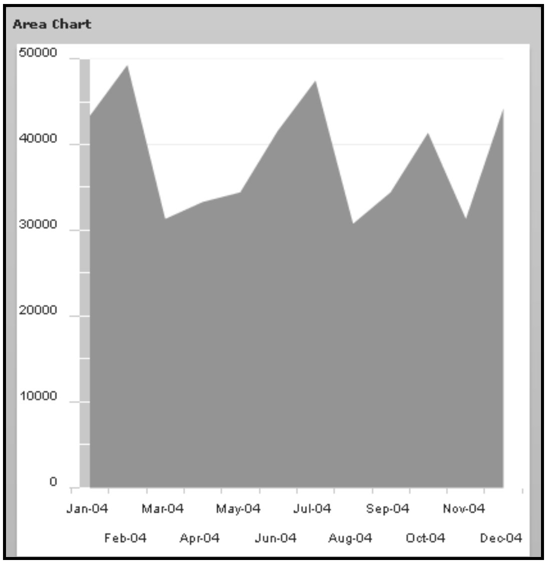
Adobe Flex provides different types of charting controls.

## Contents

## Using area charts

You use the AreaChart control to represent data as an area bounded by a line connecting the values in the data. The area underneath the line is filled in with a color or pattern. You can use an icon or symbol to represent each data point along the line, or you can show a simple line without icons.

The following image shows an example of an area chart:



The following example creates an AreaChart control:

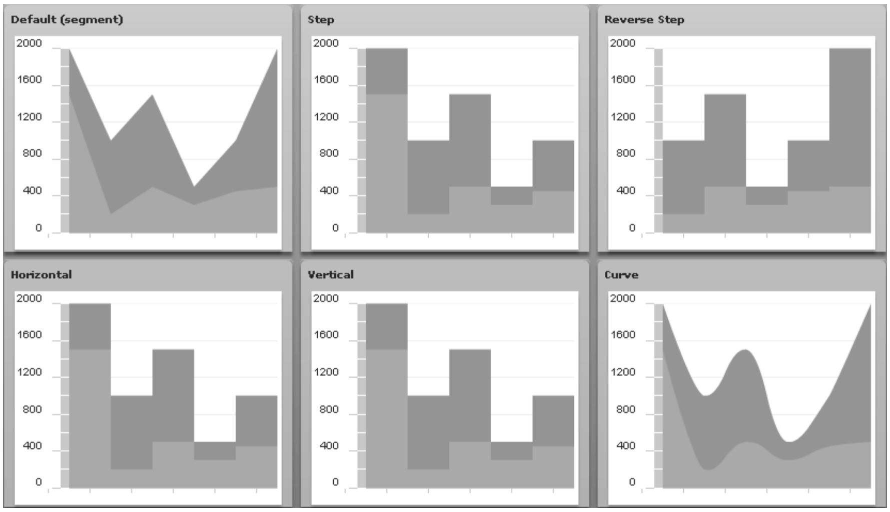
```
<?xml version="1.0"?>
<!-- charts/BasicArea.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Area Chart">
     <mx:AreaChart id="myChart" dataProvider="{expenses}"
     showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:AreaSeries
                yField="Profit"
                displayName="Profit"
           />
           <mx:AreaSeries
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:AreaChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

An area chart is essentially a line chart with the area underneath the line filled in; therefore, area charts and line charts share many of the same characteristics. For more information, see “Using line charts” on page 1673.

You use the AreaSeries chart series with the AreaChart control to define the data for the chart. The following table describes properties of the AreaSeries chart series that you commonly use to define your chart:

| Property | Description |
|---|---|
| `yField` | Specifies the field of the data provider that determines the y-axis location of each data point. |
| `xField` | Specifies the field of the data provider that determines the x-axis location of each data point. If you omit this field, Flex arranges the data points in the order of the data in the data provider. |
| `minField` | Specifies the field of the data provider that determines the y-axis location of the bottom of an area. This property is optional. If you omit it, the bottom of the area is aligned with the x-axis. This property has no effect on overlaid, stacked, or 100% stacked charts. For more information on using the `minField` property, see "Using the minField property" on page 1865. |
| `form` | Specifies the way in which the data series is shown in the chart. The following values are valid: <br>• `segment` Draws lines as connected segments that are angled to connect at each data point in the series. This is the default. <br>• `step` Draws lines as horizontal and vertical segments. At the first data point, draws a horizontal line, and then a vertical line to the second point. Repeats this action for each data point. <br>• `reverseStep` Draws lines as horizontal and vertical segments. At the first data point, draws a vertical line, and then a horizontal line to the second point. Repeats this action for each data point. <br>• `vertical` Draws only the vertical line from the *y*-coordinate of the first point to *y*-coordinate of the second point at the *x*-coordinate of the second point. Repeats this action for each data point. <br>• `horizontal` Draws only the horizontal line from the *x*-coordinate of the first point to x-coordinate of the second point at the *y*-coordinate of the first point. Repeats this action for each data point. <br>• `curve` Draws curves between data points. |

The following example shows the available forms for an AreaSeries series:



You can use the `type` property of the AreaChart control to represent a number of chart variations, including overlaid, stacked, 100% stacked, and high-low areas. For more information, see "Stacking charts" on page 1867.

## Using bar charts

You use the BarChart control to represent data as a series of horizontal bars whose length is determined by values in the data. You can use the BarChart control to represent a number of chart variations, including clustered bars, overlaid, stacked, 100% stacked, and high-low areas. For more information, see "Stacking charts" on page 1867.

You use the BarSeries chart series with the BarChart control to define the data for the chart. The following table describes the properties of the BarSeries chart series that you use to define your chart:

| Property | Description |
|---|---|
| `yField` | Specifies the field of the data provider that determines the y-axis location of the base of each bar in the chart. If you omit this property, Flex arranges the bars in the order of the data in the data provider. |

| Property | Description |
|---|---|
| `xField` | Specifies the field of the data provider that determines the x-axis location of the end of each bar. |
| `minField` | Specifies the field of the data provider that determines the x-axis location of the base of a bar. This property has no effect in overlaid, stacked, or 100% stacked charts. For more information on using the `minField` property, see "Using the minField property" on page 1865. |

The following example creates a simple BarChart control:

```
<?xml version="1.0"?>
<!-- charts/BasicBar.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Bar Chart">
     <mx:BarChart id="myChart" dataProvider="{expenses}">
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries
                yField="Month"
                xField="Profit"
                displayName="Profit"
           />
           <mx:BarSeries
                yField="Month"
                xField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

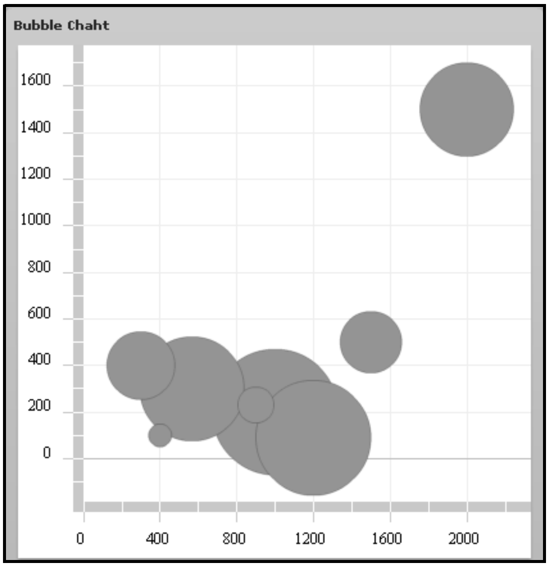
A bar chart is essentially a column chart rotated 90 degrees clockwise; therefore, bar charts and column charts share many of the same characteristics. For more information, see "Using column charts" on page 1662.

## Using bubble charts

You use the BubbleChart control to represent data with three values for each data point: a value that determines its position along the x-axis, a value that determines its position along the y-axis, and a value that determines the size of the chart symbol, relative to the other data points on the chart.

The `<mx:BubbleChart>` tag takes an additional property, `maxRadius`. This property specifies the maximum radius of the largest chart element, in pixels. The data point with the largest value is assigned this radius; all other data points are assigned a smaller radius based on their value relative to the largest value. The default value is 30 pixels.

The following example shows a bubble chart:



You use the BubbleSeries chart series with the BubbleChart control to define the data for the chart. The following table describes the properties of the BubbleSeries chart series that you commonly use to define your chart:

| Property | Description |
|---|---|
| `yField` | Specifies the field of the data provider that determines the y-axis location of each data point. This property is required. |
| `xField` | Specifies the field of the data provider that determines the x-axis location of each data point. This property is required. |
| `radiusField` | Specifies the field of the data provider that determines the radius of each symbol, relative to the other data points in the chart. This property is required. |

The following example draws a BubbleChart control and sets the maximum radius of bubble elements to 50:

```
<?xml version="1.0"?>
<!-- charts/BasicBubble.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:120, Amount:45},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:60},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:30}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Bubble Chart">
     <mx:BubbleChart id="myChart"
        maxRadius="50"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:series>
           <mx:BubbleSeries
                xField="Profit"
                yField="Expenses"
                radiusField="Amount"
                displayName="Profit"
           />
        </mx:series>
     </mx:BubbleChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

## Using multiple series in BubbleChart controls

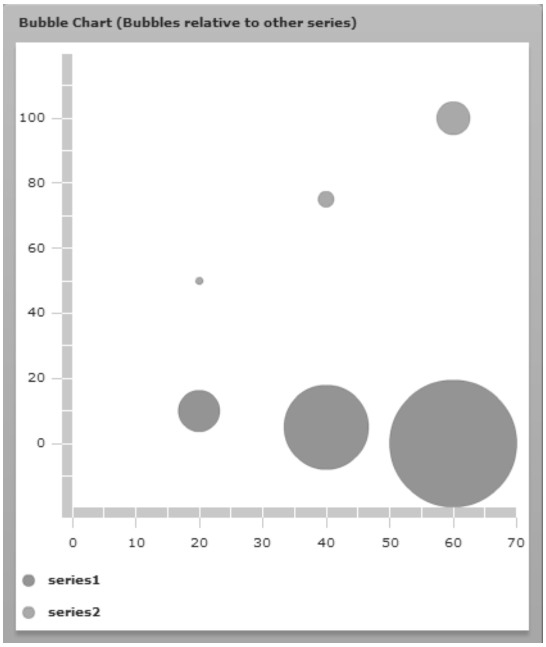
As with other chart controls, you can have multiple series in a BubbleChart control. There is an additional consideration when using multiple series in a BubbleChart control. You must decide whether you want the size of the bubbles in both series to be relative to bubbles in the other series or relative to only the other bubbles in their own series.

For example, you have two series, A and B. Series A has bubbles with radius values of 10, 20, and 30. Series B has bubbles with radius values of 2, 4, and 8. Your BubbleChart control can display bubbles in series A that are all larger than the bubbles in series B. You can also design a BubbleChart control so that the bubbles' sizes in series A are not relative to bubbles in series B. Flex renders the bubble with a radius 10 in series A and the bubble with a radius of 1 in series B the same size.

If you want the size of the bubbles to be relative to each other in the different series, add both series in the series array, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/BubbleRelativeSize.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;
            [Bindable]
            private var s1:ArrayCollection = new ArrayCollection( [
                {"x": 20, "y": 10, "r":10 },
                {"x": 40, "y": 5, "r":20 } ,
                {"x": 60, "y": 0, "r":30 }]);

            [Bindable]
            private var s2:ArrayCollection = new ArrayCollection( [
                {"x": 20, "y": 50, "r":2 },
                {"x": 40, "y": 75, "r":4 },
                {"x": 60, "y": 100, "r":8 } ]);

    ]]>
    </mx:Script>

    <!-- Empty Array to clear the background grid lines. -->
    <mx:Array id="bge">
    </mx:Array>

    <mx:Panel title="Bubble Chart (Bubbles relative to other series)">
        <mx:BubbleChart id="myChart"
            showDataTips="true"
            backgroundElements="{bge}"
        >
            <mx:series>
                <mx:BubbleSeries
                    dataProvider="{s1}"
                    displayName="series1"
                    xField="x"
                    yField="y"
                    radiusField="r"
                />
                <mx:BubbleSeries
                    dataProvider="{s2}"
                    displayName="series2"
                    xField="x"
                    yField="y"
                    radiusField="r"
                />
            </mx:series>
        </mx:BubbleChart>
        <mx:Legend dataProvider="{myChart}"/>
    </mx:Panel>
</mx:Application>
```

The following example shows a BubbleChart control with two series whose bubbles' sizes are relative to one another:



```
<?xml version="1.0"?>
<!-- charts/BubbleNonRelativeSize.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;
            [Bindable]
            private var s1:ArrayCollection = new ArrayCollection( [
                {"x": 20, "y": 10, "r":10 },
                {"x": 40, "y": 5, "r":20 } ,
                {"x": 60, "y": 0, "r":30 }]);

            [Bindable]
            private var s2:ArrayCollection = new ArrayCollection( [
                {"x": 20, "y": 50, "r":2 },
                {"x": 40, "y": 75, "r":4 },
                {"x": 60, "y": 100, "r":8 } ]);

    ]]>
    </mx:Script>

    <!-- Empty Array to clear the background grid lines. -->
    <mx:Array id="bge">
    </mx:Array>

    <mx:Panel title="Bubble Chart (Bubbles not relative across series)">
        <mx:BubbleChart id="myChart"
            showDataTips="true"
            dataProvider="{s1}"
            secondDataProvider="{s2}"
            backgroundElements="{bge}"
        >
            <mx:series>
                <mx:BubbleSeries
                    dataProvider="{s1}"
                    displayName="series1"
                    xField="x"
                    yField="y"
                    radiusField="r"
                />
            </mx:series>
            <mx:secondSeries>
                <mx:BubbleSeries
                    dataProvider="{s2}"
                    displayName="series2"
                    xField="x"
                    yField="y"
                    radiusField="r"
                />
            </mx:secondSeries>
        </mx:BubbleChart>
        <mx:Legend dataProvider="{myChart}"/>
    </mx:Panel>
```

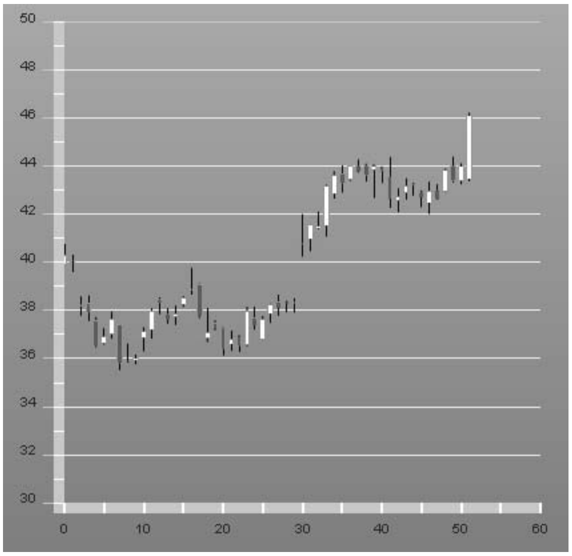
## `</mx:Application>`Using candlestick charts

The CandlestickChart control represents financial data as a series of candlesticks representing the high, low, opening, and closing values of a data series. The top and bottom of the vertical line in each candlestick represent the high and low values for the data point, while the top and bottom of the filled box represents the opening and closing values. Each candlestick is filled differently depending on whether the closing value for the data point is higher or lower than the opening value.

The CandlestickChart control's CandlestickSeries requires all four data points: high, low, open, and close. If you do not want to use opening value data points, you can use the HighLowOpenClose charts, which do not require a data point that represents the opening value. For more information, see "Using HighLowOpenClose charts" on page 1669.

You use the CandlestickSeries chart series with the CandlestickChart control to define the data.

The following shows an example of a CandlestickChart chart:



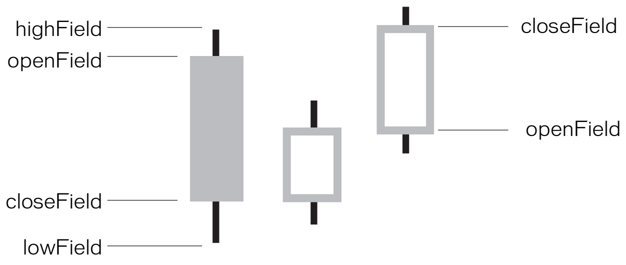
The following table describes the properties of the CandlestickSeries chart series that you commonly use to define your chart:

| Property | Description |
|---|---|
| `closeField` | Specifies the field of the data provider that determines the y-axis location of the closing value of the element. This property defines the top or bottom of the candlestick. |
| `highField` | Specifies the field of the data provider that determines the y-axis location of the high value of the element. This property defines the top of the line inside the candlestick. |
| `lowField` | Specifies the field of the data provider that determines the y-axis location of the low value of the element. This property defines the bottom of the line inside the candlestick. |

| Property | Description |
|---|---|
| `openField` | Specifies the field of the data provider that determines the y-axis location of the opening value of the element. This property defines the position of the top or bottom of the candlestick. |
| `xField` | Specifies the field of the data provider that determines the x-axis location of the element. If set to the empty string (""), Flex renders the columns in the order in which they appear in the data provider. The default value is the empty string. |

If the `closeField` is lower than the `openField`, Flex applies a solid fill to the candle. The color of this solid fill defaults to the color of the box's outline. It is defined by the `declineFill` style property. If the `closeField` is *higher* than the `openField`, Flex fills the candle with white by default.

The following image shows these properties. As you can see, the location of the `closeField` property can be either the top or the bottom of the candlestick, depending on whether it is higher or lower than the `openField` property:



The following example creates a CandlestickChart control:

```
<?xml version="1.0"?>
<!-- charts/BasicCandlestick.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     [Bindable]
     public var TICKER:Array = [
        {date:"1-Aug-05",open:42.57,high:43.08,low:42.08,close:42.75},
        {date:"2-Aug-05",open:42.89,high:43.5,low:42.61,close:43.19},
        {date:"3-Aug-05",open:43.19,high:43.31,low:42.77,close:43.22},
        {date:"4-Aug-05",open:42.89,high:43,low:42.29,close:42.71},
        {date:"5-Aug-05",open:42.49,high:43.36,low:42.02,close:42.99},
        {date:"8-Aug-05",open:43,high:43.25,low:42.61,close:42.65},
        {date:"9-Aug-05",open:42.93,high:43.89,low:42.91,close:43.82},
        {date:"10-Aug-05",open:44,high:44.39,low:43.31,close:43.38},
        {date:"11-Aug-05",open:43.39,high:44.12,low:43.25,close:44},
        {date:"12-Aug-05",open:43.46,high:46.22,low:43.36,close:46.1}
     ];
  ]]></mx:Script>

  <mx:Panel title="Candlestick Chart">
     <mx:CandlestickChart id="mychart"
        dataProvider="{TICKER}"
        showDataTips="true"
        height="400"
        width="400"
     >
        <mx:series>
           <mx:CandlestickSeries
                dataProvider="{TICKER}"
                openField="open"
                highField="high"
                lowField="low"
                closeField="close"
                displayName="TICKER"
           />
        </mx:series>
     </mx:CandlestickChart>
     <mx:Legend dataProvider="{mychart}"/>
  </mx:Panel>
</mx:Application>
```

You can change the color of the candle's fill with the `fill` and `declineFill` properties of the series. The `fill` property defines the color of the candlestick when the `closeField` value is higher than the `openField` value. The `declineFill` property defines the color of the candlestick when the reverse is true. You can also define the properties of the high-low lines and candlestick borders by using a Stroke class, as the following example shows:

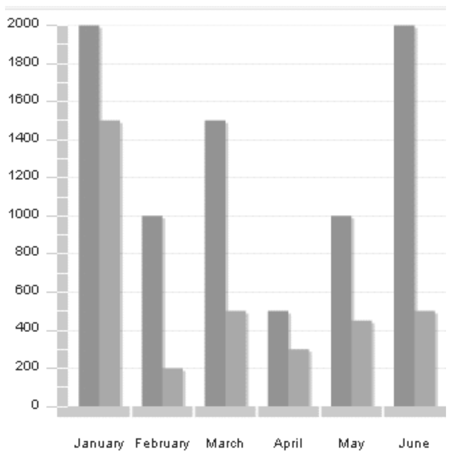
```
<?xml version="1.0"?>
<!-- charts/CandlestickStyles.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var TICKER:ArrayCollection = new ArrayCollection([
        {date:"1-Aug-05",open:42.57,high:43.08,low:42.08,close:42.75},
        {date:"2-Aug-05",open:42.89,high:43.5,low:42.61,close:43.19},
        {date:"3-Aug-05",open:43.19,high:43.31,low:42.77,close:43.22},
        {date:"4-Aug-05",open:42.89,high:43,low:42.29,close:42.71},
        {date:"5-Aug-05",open:42.49,high:43.36,low:42.02,close:42.99},
        {date:"8-Aug-05",open:43,high:43.25,low:42.61,close:42.65},
        {date:"9-Aug-05",open:42.93,high:43.89,low:42.91,close:43.82},
        {date:"10-Aug-05",open:44,high:44.39,low:43.31,close:43.38},
        {date:"11-Aug-05",open:43.39,high:44.12,low:43.25,close:44},
        {date:"12-Aug-05",open:43.46,high:46.22,low:43.36,close:46.1}
     ]);
  ]]></mx:Script>

  <mx:Panel title="Candlestick Chart">
     <mx:CandlestickChart id="mychart"
        dataProvider="{TICKER}"
        showDataTips="true"
        height="400"
        width="400"
     >
        <mx:verticalAxis>
           <mx:LinearAxis title="linear axis" minimum="40" maximum=
           "50"/>
        </mx:verticalAxis>

        <mx:series>
           <mx:CandlestickSeries
                dataProvider="{TICKER}"
                openField="open"
                highField="high"
                lowField="low"
                closeField="close"
                displayName="TICKER"
           >
            <mx:fill>
```

```
                <mx:SolidColor color="green"/>
            </mx:fill>
            <mx:declineFill>
                <mx:SolidColor color="red"/>
            </mx:declineFill>
            <mx:stroke>
                <mx:Stroke weight="1" color="black"/>
            </mx:stroke>
           </mx:CandlestickSeries>
        </mx:series>
     </mx:CandlestickChart>
     <mx:Legend dataProvider="{mychart}"/>
  </mx:Panel>
</mx:Application>
```

## Using column charts

The ColumnChart control represents data as a series of vertical columns whose height is determined by values in the data. You can use the ColumnChart control to create several variations of column charts, including simple columns, clustered columns, overlaid, stacked, 100% stacked, and high-low. For more information, see "Stacking charts" on page 1867.

The following example shows a ColumnChart control with two series:



You use the ColumnSeries chart series with the ColumnChart control to define the data for the chart. The following table describes the properties of the ColumnSeries chart series to define your chart:

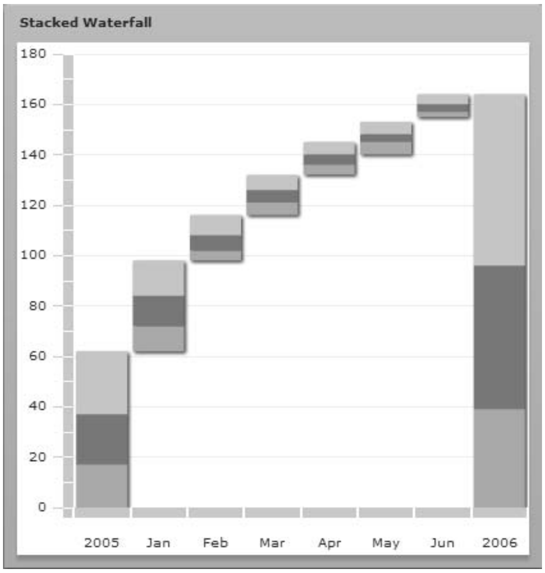
| Property | Description |
|---|---|
| `yField` | Specifies the field of the data provider that determines the y-axis location of the top of a column. This field defines the height of the column. |
| `xField` | Specifies the field of the data provider that determines the x-axis location of the column. If you omit this property, Flex arranges the columns in the order of the data in the data provider. |
| `minField` | Specifies the field of the data provider that determines the y-axis location of the bottom of a column. This property has no effect on overlaid, stacked, or 100% stacked charts. For more information on using the `minField` property, see "Using the minField property" on page 1865. |

The following example creates a ColumnChart control with two series:

```
<?xml version="1.0"?>
<!-- charts/BasicColumn.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

You can set the `type` property on a ColumnChart control to stack and group series data in the chart. For more information, see “Stacking charts” on page 1867.

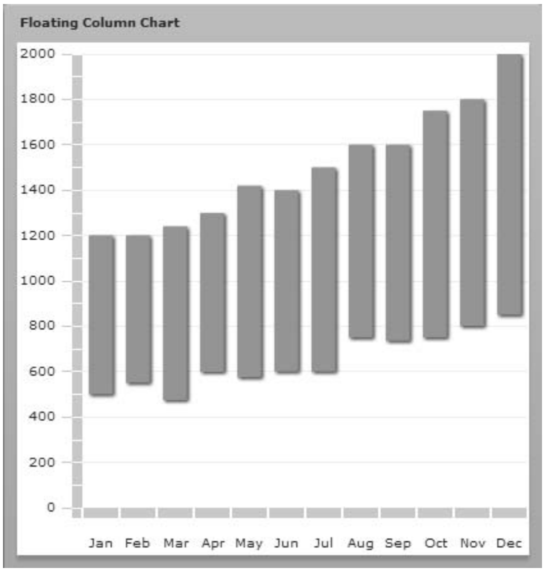
You can create cascading or waterfall column charts by using the ColumnChart control. One way to do this is to create an invisible series and use that to set the variable height of the other columns, creating the waterfall effect. The following is an example of a waterfall chart:



The following code creates this chart:

```
<?xml version="1.0"?>
<!-- charts/WaterfallStacked.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"2005", top:25, middle:20, bottom:17, Invisible:0},
        {Month:"Jan", top:14, middle:12, bottom:10, Invisible:62},
        {Month:"Feb", top:8, middle:6, bottom:4, Invisible:98},
        {Month:"Mar", top:6, middle:5, bottom:5, Invisible:116},
        {Month:"Apr", top:5, middle:4, bottom:4, Invisible:132},
        {Month:"May", top:5, middle:3, bottom:5, Invisible:140},
        {Month:"Jun", top:4, middle:3, bottom:2, Invisible:155},
        {Month:"2006", top:68, middle:57, bottom:39, Invisible:0}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Stacked Waterfall">
     <mx:ColumnChart id="myChart"
        dataProvider="{expenses}"
        columnWidthRatio=".9"
        showDataTips="true"
        type="stacked"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
            />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                yField="Invisible"
                displayName="Invisible"
           >
                <mx:fill>
                    <!--Set alpha to 0 to hide invisible column.-->
                    <mx:SolidColor color="0xFFFFFF" alpha="0"/>
                </mx:fill>
           </mx:ColumnSeries>
           <mx:ColumnSeries yField="bottom" displayName="Profit"/>
           <mx:ColumnSeries yField="middle" displayName="Expenses"/>
           <mx:ColumnSeries yField="top" displayName="Profit"/>
        </mx:series>
     </mx:ColumnChart>
  </mx:Panel>
</mx:Application>
```

You can also create floating column charts by using the `minField` property of the chart's data series. This property lets you set the lower level of a column. The following example shows a floating ColumnChart control:



The following code draws this chart:

```
<?xml version="1.0"?>
<!-- charts/MinFieldColumn.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Revenue:1200, Expenses:500},
        {Month:"Feb", Revenue:1200, Expenses:550},
        {Month:"Mar", Revenue:1240, Expenses:475},
        {Month:"Apr", Revenue:1300, Expenses:600},
        {Month:"May", Revenue:1420, Expenses:575},
        {Month:"Jun", Revenue:1400, Expenses:600},
        {Month:"Jul", Revenue:1500, Expenses:600},
        {Month:"Aug", Revenue:1600, Expenses:750},
        {Month:"Sep", Revenue:1600, Expenses:735},
        {Month:"Oct", Revenue:1750, Expenses:750},
        {Month:"Nov", Revenue:1800, Expenses:800},
        {Month:"Dec", Revenue:2000, Expenses:850}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Floating Column Chart">
     <mx:ColumnChart
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
            />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                yField="Revenue"
                minField="Expenses"
                displayName="Revenue"
            />
        </mx:series>
     </mx:ColumnChart>
  </mx:Panel>
</mx:Application>
```

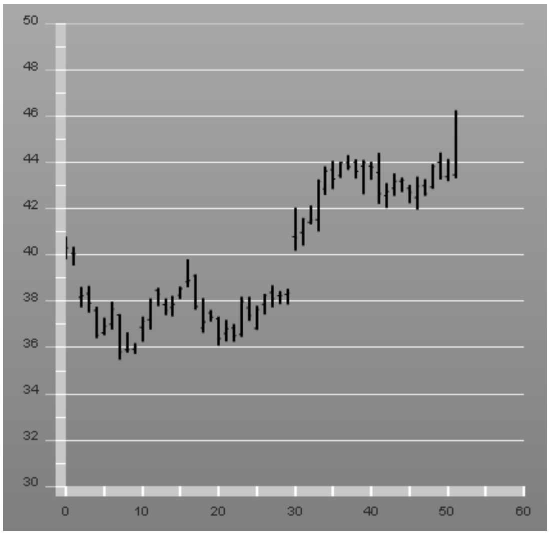
For more information, see "Using the minField property" on page 1865.

## Using HighLowOpenClose charts

The HLOCChart control represents financial data as a series of lines representing the high, low, opening, and closing values of a data series. The top and bottom of the vertical line represent the high and low values for the data point, while the left tick mark represents the opening values and the right tick mark represents the closing values.

The HLOCChart control does not require a data point that represents the opening value. A related chart is the CandlestickChart control that represents similar data as candlesticks. For more information see "</mx:Application>Using candlestick charts" on page 1657.

You use the HLOCSeries with the HLOCChart control to define the data for HighLowOpenClose charts. The following example shows an HLOC chart:

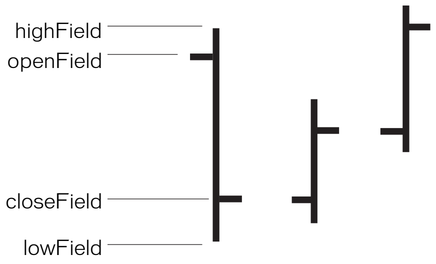


The following table describes the properties of the HLOCChart control's series that you commonly use to define your chart:

| Property | Description |
|---|---|
| `closeField` | Specifies the field of the data provider that determines the *y*-axis location of the closing value of the element. This property defines the position of the right tick mark on the vertical line. |
| `highField` | Specifies the field of the data provider that determines the *y*-axis location of the high value of the element. This property defines the top of the vertical line. |

| Property | Description |
|---|---|
| `lowField` | Specifies the field of the data provider that determines the *y*-axis location of the low value of the element. This property defines the bottom of the vertical line. |
| `openField` | Specifies the field of the data provider that determines the y-axis location of the opening value of the element. This property defines the position of the left tick mark on the vertical line. This property is optional. |
| `xField` | Specifies the field of the data provider that determines the x-axis location of the element. If set to the empty string (""), Flex renders the columns in the order in which they appear in the data provider. The default value is the empty string. |

Data points in an HLOCChart control do not require an `openField` property. If no `openField` property is specified, Flex renders the data point as a flat line with a single closing value indicator pointing to the right. If an `openField` property is specified, Flex renders the data point with another indicator pointing to the left, as the following image shows:



The following example creates an HLOCChart control:

```
<?xml version="1.0"?>
<!-- charts/BasicHLOC.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var TICKER:ArrayCollection = new ArrayCollection([
        {date:"1-Aug-05",open:42.57,high:43.08,low:42.08,close:42.75},
        {date:"2-Aug-05",open:42.89,high:43.5,low:42.61,close:43.19},
        {date:"3-Aug-05",open:43.19,high:43.31,low:42.77,close:43.22},
        {date:"4-Aug-05",open:42.89,high:43,low:42.29,close:42.71},
        {date:"5-Aug-05",open:42.49,high:43.36,low:42.02,close:42.99},
        {date:"8-Aug-05",open:43,high:43.25,low:42.61,close:42.65},
        {date:"9-Aug-05",open:42.93,high:43.89,low:42.91,close:43.82},
        {date:"10-Aug-05",open:44,high:44.39,low:43.31,close:43.38},
        {date:"11-Aug-05",open:43.39,high:44.12,low:43.25,close:44},
        {date:"12-Aug-05",open:43.46,high:46.22,low:43.36,close:46.1}
     ]);
  ]]></mx:Script>

  <mx:Panel title="HighLowOpenClose Chart">
     <mx:HLOCChart id="myChart"
        dataProvider="{TICKER}"
        showDataTips="true"
     >
        <mx:verticalAxis>
           <mx:LinearAxis minimum="30" maximum="50"/>
        </mx:verticalAxis>
        <mx:series>
           <mx:HLOCSeries
                dataProvider="{TICKER}"
                openField="open"
                highField="high"
                lowField="low"
                closeField="close"
                displayName="TICKER"
            >
           </mx:HLOCSeries>
        </mx:series>
     </mx:HLOCChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

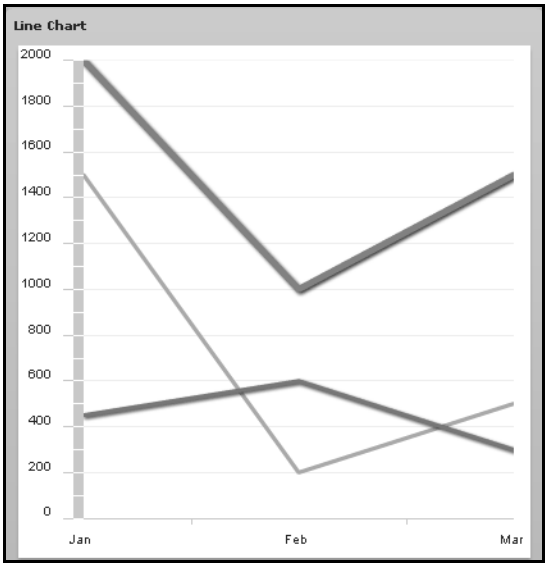
You can change the stroke of the vertical lines by using a Stroke object on the series. To change the appearance of the opening and closing value tick marks on the vertical line, you use the `openTickStroke` and `closeTickStroke` style properties. The following example changes the stroke of the vertical line to 2 (the default value is 1) and the color of all the lines to black:

```
<?xml version="1.0"?>
<!-- charts/HLOCStyled.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var TICKER:ArrayCollection = new ArrayCollection([
        {date:"1-Aug-05",open:42.57,high:43.08,low:42.08,close:42.75},
        {date:"2-Aug-05",open:42.89,high:43.5,low:42.61,close:43.19},
        {date:"3-Aug-05",open:43.19,high:43.31,low:42.77,close:43.22},
        {date:"4-Aug-05",open:42.89,high:43,low:42.29,close:42.71},
        {date:"5-Aug-05",open:42.49,high:43.36,low:42.02,close:42.99},
        {date:"8-Aug-05",open:43,high:43.25,low:42.61,close:42.65},
        {date:"9-Aug-05",open:42.93,high:43.89,low:42.91,close:43.82},
        {date:"10-Aug-05",open:44,high:44.39,low:43.31,close:43.38},
        {date:"11-Aug-05",open:43.39,high:44.12,low:43.25,close:44},
        {date:"12-Aug-05",open:43.46,high:46.22,low:43.36,close:46.1},
     ]);
  ]]></mx:Script>

  <mx:Panel title="HLOC Chart">
     <mx:HLOCChart id="myChart"
        dataProvider="{TICKER}"
        showDataTips="true"
     >
        <mx:verticalAxis>
           <mx:LinearAxis minimum="30" maximum="50"/>
        </mx:verticalAxis>
        <mx:series>
           <mx:HLOCSeries
                dataProvider="{TICKER}"
                openField="open"
                highField="high"
                lowField="low"
                closeField="close"
                displayName="TICKER"
            >
            <mx:stroke>
                <mx:Stroke color="#000000" weight="2"/>
            </mx:stroke>
            <mx:closeTickStroke>
                <mx:Stroke color="#000000" weight="1"/>
            </mx:closeTickStroke>
            <mx:openTickStroke>
                <mx:Stroke color="#000000" weight="1"/>
            </mx:openTickStroke>
           </mx:HLOCSeries>
        </mx:series>
     </mx:HLOCChart>
     </mx:Panel>
     <mx:Legend dataProvider="{myChart}"/>
</mx:Application>
```

## Using line charts

The LineChart control represents data as a series of points, in Cartesian coordinates, connected by a continuous line. You can use an icon or symbol to represent each data point along the line, or show a simple line without icons.

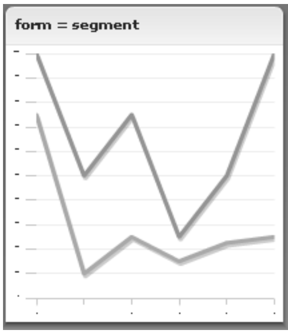
The following example shows a simple line chart:



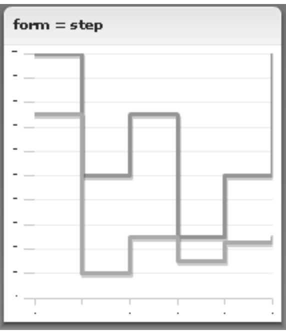
You use the LineSeries chart series with the LineChart control to define the data for the chart. The following table describes the properties of the LineSeries chart series you commonly use to define your chart:

| Property | Description |
|---|---|
| `yField` | Specifies the field of the data provider that determines the y-axis location of each data point. This is the height of the line at that location along the axis. |
| `xField` | Specifies the field of the data provider that determines the x-axis location of each data point. If you omit this field, Flex arranges the data points in the order of the data in the data provider. |
| `interpolateValues` | Specifies how to represent missing data. If you set the value of this property to `false`, the chart breaks the line at the missing value. If you specify a value of `true`, Flex draws a continuous line by interpolating the missing value. The default value is `false`. |
| `form` | Specifies the way in which the data series is shown in the chart. The following values are valid:<br>• `segment` Draws lines as connected segments that are angled to connect at each data point in the series. This is the default.<br>• `step` Draws lines as horizontal and vertical segments. At the first data point, draws a horizontal line, and then a vertical line to the second point. Repeats this for each data point.<br>• `reverseStep` Draws lines as horizontal and vertical segments. At the first data point, draws a vertical line, and then a horizontal line to the second point. Repeats this for each data point.<br>• `vertical` Draws the vertical line only from the *y*-coordinate of the first point to the *y*-coordinate of the second point at the *x*-coordinate of the second point. Repeats this for each data point.<br>• `horizontal` Draws the horizontal line only from the *x*-coordinate of the first point to the *x*-coordinate of the second point at the *y*-coordinate of the first point. Repeats this for each data point.<br>• `curve` Draws curves between data points. |

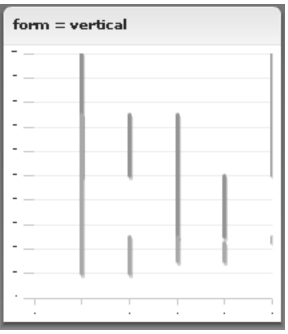
The following example shows the available forms for a LineSeries chart:



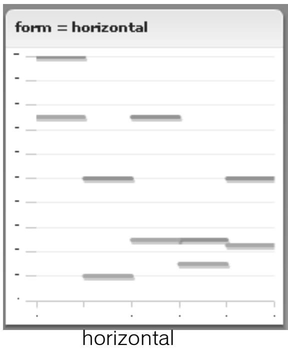
segment



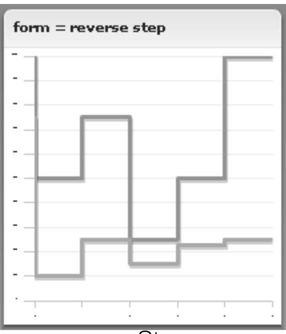
step



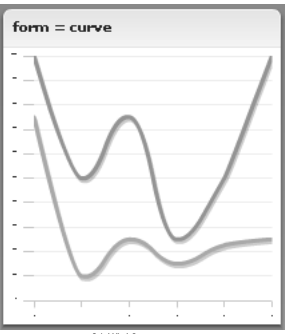
vertical



horizontal



reverseStep



curve

The following example creates a LineChart control:

```
<?xml version="1.0"?>
<!-- charts/BasicLine.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Line Chart">
     <mx:LineChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
            />
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries
                yField="Profit"
                displayName="Profit"
           />
           <mx:LineSeries
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:LineChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

## Formatting lines

You can change the width and color of the lines for each series by using the `<mx:lineStroke>` tag. The default line is 3 pixels wide and has a shadow. The following example sets a custom color and width for the series Stroke object:

```
<?xml version="1.0"?>
<!-- charts/BasicLineStroke.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Line Chart With Strokes">
     <mx:LineChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
            />
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries
            yField="Profit"
            displayName="Profit"
           >
            <mx:lineStroke>
                <mx:Stroke
                    color="0x0099FF"
                    weight="20"
                    alpha=".2"
                />
            </mx:lineStroke>
           </mx:LineSeries>
           <mx:LineSeries
            yField="Expenses"
            displayName="Expenses"
           >
            <mx:lineStroke>
                <mx:Stroke
                    color="0x0044EB"
                    weight="20"
                    alpha=".8"
                />
            </mx:lineStroke>
           </mx:LineSeries>
        </mx:series>
     </mx:LineChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

For more information on using the Stroke class in charts, see “Using strokes” on page 1733.

The default appearance of the lines in a LineChart control is with drop shadows. You can remove these shadows by setting the chart control's `seriesFilters` property to an empty Array, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/LineChartNoShadows.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);
  ]]></mx:Script>

  <mx:Panel title="Line Chart with No Shadows">
     <mx:LineChart id="myChart" dataProvider="{expenses}">
        <mx:seriesFilters>
           <mx:Array/>
        </mx:seriesFilters>
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries
                yField="Profit"
                displayName="Profit"
           />
           <mx:LineSeries
                yField="Expenses"
                displayName="Expenses"
           />
           <mx:LineSeries
                yField="Amount"
                displayName="Amount"
           />
        </mx:series>
     </mx:LineChart>
  </mx:Panel>
</mx:Application>
```

You can also set the value of the `seriesFilters` property programmatically, as the following example shows:

```
myLineChart.seriesFilters = [];
```

You can also specify a programmatic renderer (or *skin*) class for each series by setting the `lineSegmentRenderer` property of the LineSeries. The default renderer is the LineRenderer, but Flex also applies a shadow filter on all series. If you remove the shadow filter, as the previous example shows, but want a line with a drop shadow in your chart, you can set the `lineSegmentRenderer` to the ShadowLineRenderer class, as the following example shows:

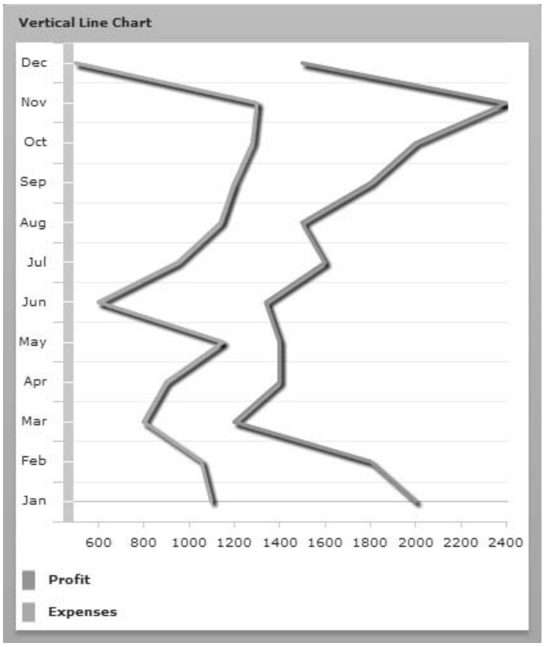
```
<?xml version="1.0"?>
<!-- charts/LineChartOneShadow.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);
  ]]></mx:Script>

  <mx:Panel title="Line Chart with One Shadow">
     <mx:LineChart id="myChart" dataProvider="{expenses}">
        <mx:seriesFilters>
           <mx:Array/>
        </mx:seriesFilters>
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries
                yField="Profit"
                displayName="Profit"
           />
           <mx:LineSeries
                yField="Expenses"
                displayName="Expenses"
           />
           <mx:LineSeries
                yField="Amount"
                displayName="Amount"
                lineSegmentRenderer=
                "mx.charts.renderers.ShadowLineRenderer"
            />
        </mx:series>
     </mx:LineChart>
  </mx:Panel>
```

```
</mx:Application>
```

For more information on using renderer classes to change the appearance of ChartItem objects such as the LineChart control's line segments, see "Skinning ChartItem objects" on page 1785.

## Using vertical lines in a LineChart control

You can create LineChart controls that show vertical progression. The following example shows two LineSeries in a LineChart control that are displayed vertically rather than horizontally:



To make lines in a LineChart control display vertically rather than horizontally, you must do the following:

- Explicitly define the `xField` and `yField` properties for the LineSeries object.
- Set the `sortOnXField` property of the LineSeries object to `false`.

By default, data points in a series are sorted from left to right (on the x-axis) before rendering. This causes the LineSeries to draw horizontally. When you disable the `xField` sort and explicitly define a `yField` property, Flex draws the lines vertically rather than horizontally.

Flex does not sort any data vertically. As a result, you must ensure that your data is in order in the data provider. If it is not in order, Flex renders a zig-zagging line up and down the chart as it connects that dots according to position in the data provider.

The following example creates a LineChart control that displays vertical lines:

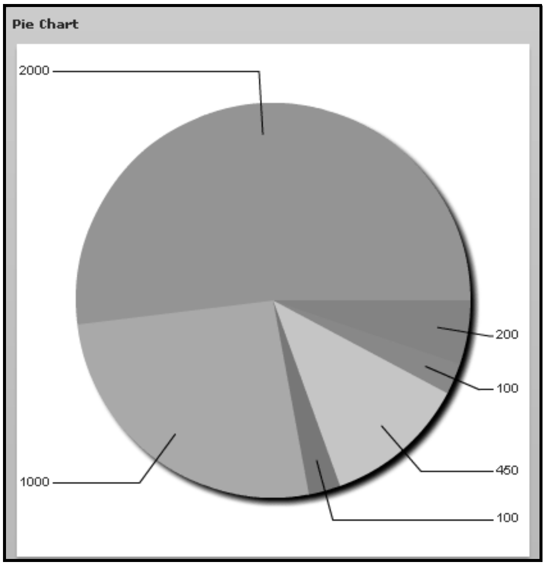
```
<?xml version="1.0"?>
<!-- charts/VerticalLineChart.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1100},
        {Month:"Feb", Profit:1800, Expenses:1055},
        {Month:"Mar", Profit:1200, Expenses:800},
        {Month:"Apr", Profit:1400, Expenses:900},
        {Month:"May", Profit:1400, Expenses:1150},
        {Month:"Jun", Profit:1340, Expenses:600},
        {Month:"Jul", Profit:1600, Expenses:950},
        {Month:"Aug", Profit:1500, Expenses:1140},
        {Month:"Sep", Profit:1800, Expenses:1200},
        {Month:"Oct", Profit:2000, Expenses:1280},
        {Month:"Nov", Profit:2400, Expenses:1300},
        {Month:"Dec", Profit:1500, Expenses:500}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Vertical Line Chart">
     <mx:LineChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:verticalAxis>
        <mx:series>
           <mx:LineSeries
                xField="Profit"
                yField="Month"
                displayName="Profit"
                sortOnXField="false"
           />
           <mx:LineSeries
                xField="Expenses"
                yField="Month"
                displayName="Expenses"
                sortOnXField="false"
           />
        </mx:series>
     </mx:LineChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
```

```
</mx:Application>
```

## Using pie charts

You use the PieChart control to define a standard pie chart. The data for the data provider determines the size of each wedge in the pie chart relative to the other wedges.

The following example shows a pie chart:



You use the PieSeries chart series with the PieChart control to define the data for the chart. The PieSeries can create standard pie charts or doughnut charts. PieChart controls also support labels that identify data points.

The following table describes the properties of the PieChart control's PieSeries chart series that you commonly use to define your chart:

| Property | Description |
|---|---|
| `field` | Specifies the field of the data provider that determines the data for each wedge of the pie chart. |
| `labelPosition` | Specifies how to render data labels for the wedges. |
| `nameField` | Specifies the field of the data provider to use as the name for the wedge in DataTips and legends. |

The following example defines a PieChart control:

```
<?xml version="1.0"?>
<!-- charts/BasicPie.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Expense:"Taxes", Amount:2000},
        {Expense:"Rent", Amount:1000},
        {Expense:"Bills", Amount:100},
        {Expense:"Car", Amount:450},
        {Expense:"Gas", Amount:100},
        {Expense:"Food", Amount:200}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Pie Chart">
     <mx:PieChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:series>
           <mx:PieSeries
                field="Amount"
                nameField="Expense"
                labelPosition="callout"
           />
        </mx:series>
     </mx:PieChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

## Using data labels with PieChart controls

PieChart controls support data labels that display information about each data point. All charts support DataTips, which display the value of a data point when the user moves the mouse over it. Data labels, on the other hand, are only supported by PieSeries, ColumnSeries, and BarSeries objects. Data labels are different from DataTips in that they are always visible and do not react to mouse movements.

To add data labels to your PieChart control, set the `labelPosition` property on the series to a valid value other than `none`. To remove labels from your pie chart, set the `labelPosition` property to `none`. The default value is `none`.

The following table describes the valid values of the `labelPosition` property for a PieSeries object. The PieSeries class supports more values for this property than the BarSeries and ColumnSeries classes.

| Value | Description |
|---|---|
| `callout` | Draws labels in two vertical stacks on either side of the PieChart control. Shrinks the PieChart if necessary to make room for the labels. Draws key lines from each label to the associated wedge. Shrinks labels as necessary to fit the space provided.<br>This property can only be used with a PieSeries object. You cannot use callouts with BarSeries and ColumnSeries objects. |
| `inside` | Draws labels inside the chart. Shrinks labels to ensure that they do not overlap each other. Any label that must be drawn too small, as defined by the `insideLabelSizeLimit` property, is hidden from view. |
| `insideWithCallout` | Draws labels inside the pie, but if labels are shrunk below a legible size, Flex converts them to callout labels. This is a common value to set `labelPosition` to when the actual size of your chart is flexible and users might resize it.<br>This property can only be used with a PieSeries object. You cannot use callouts with BarSeries and ColumnSeries objects. |
| `none` | Does not draw labels. This is the default value. |
| `outside` | Draws labels around the boundary of the PieChart control. |

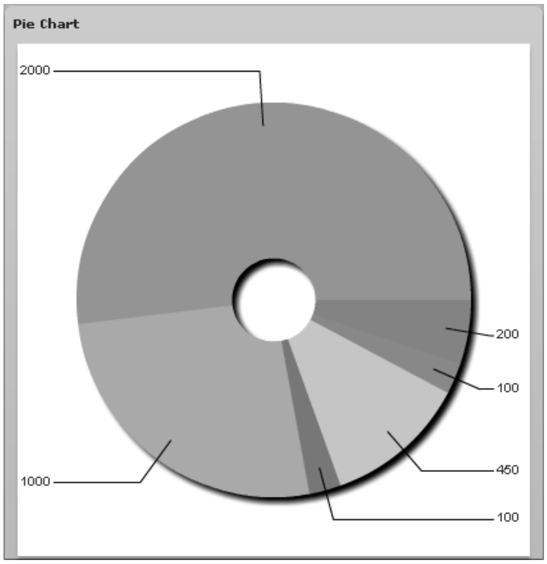
The following table describes the properties of the PieSeries object that you can use to manipulate the appearance of labels:

| Property | Description |
|---|---|
| `calloutGap` | Defines how much space, in pixels, to insert between the edge of the pie and the data labels when rendering callouts. The default value is 10 pixels. |
| `calloutStroke` | Defines the line style used to draw the lines to callouts. For more information on defining line data points, see "Using strokes" on page 1733. |
| `insideLabelSizeLimit` | Defines the size threshold, expressed in points, below which inside data labels are considered illegible. Below this threshold, data labels are either removed entirely or turned into callouts based on the setting of the series `labelPosition` property. |

You can change the value of the data labels by using the `labelFunction` property of the PieSeries object to specify a callback function. For more information, see "Customizing data labels for PieSeries objects" on page 1841.

## Creating doughnut charts

Flex lets you create doughnut charts out of PieChart controls. Doughnut charts are identical to pie charts, except that they have hollow centers and resemble wheels rather than filled circles. The following example shows a doughnut chart:



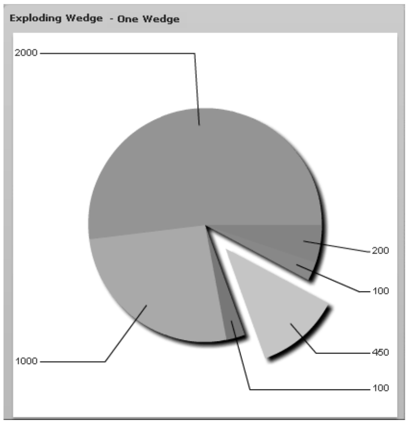
To create a doughnut chart, specify the `innerRadius` property on the PieChart control, as the following example shows:

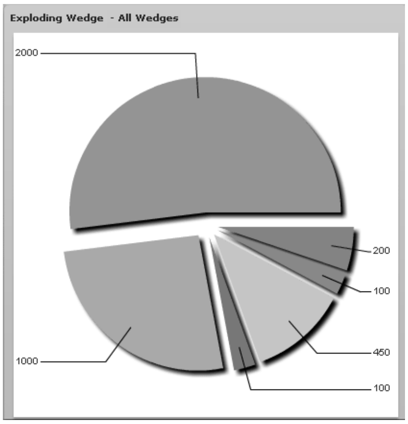
```
<?xml version="1.0"?>
<!-- charts/DoughnutPie.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Expense:"Taxes", Amount:2000},
        {Expense:"Rent", Amount:1000},
        {Expense:"Bills", Amount:100},
        {Expense:"Car", Amount:450},
        {Expense:"Gas", Amount:100},
        {Expense:"Food", Amount:200}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Pie Chart">
     <mx:PieChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
        innerRadius=".3"
     >
        <mx:series>
           <mx:PieSeries
                field="Amount"
                nameField="Expense"
                labelPosition="callout"
           />
        </mx:series>
     </mx:PieChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

The value of the `innerRadius` property is a percentage value of the “hole” compared to the entire pie’s radius. Valid values range from 0 to 1.

## Creating exploding pie charts

The PieSeries chart series supports exploding wedges, both uniformly and on a per-wedge basis, so that you can achieve effects similar to the following:





The following table describes the properties that support exploding pie charts:

| Property | Description |
|---|---|
| `explodeRadius` | A value from 0 to 1, representing the percentage of the available pie radius to use when exploding the wedges of the pie. |
| `perWedgeExplodeRadius` | An array of values from 0 to 1. The *N*th value in this array is added to the value of `explodeRadius` to determine the explode amount of each individual wedge of the pie. Individual values can be left undefined, in which case the wedge will only explode according to the `explodeRadius` property. |
| `reserveExplodeRadius` | A value from 0 to 1, representing an amount of the available pie radius to reserve for animating an exploding wedge. |

To explode all wedges of a pie chart evenly, you use the `explodeRadius` property on the PieSeries, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/ExplodingPie.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Expense:"Taxes", Amount:2000},
        {Expense:"Rent", Amount:1000},
        {Expense:"Bills", Amount:100},
        {Expense:"Car", Amount:450},
        {Expense:"Gas", Amount:100},
        {Expense:"Food", Amount:200}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Exploding Pie Chart">
     <mx:PieChart id="pie"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:series>
           <!--explodeRadius is a number between 0 and 1.-->
           <mx:PieSeries
                field="Amount"
                nameField="Expense"
                explodeRadius=".12"
           />
        </mx:series>
     </mx:PieChart>
     <mx:Legend dataProvider="{pie}"/>
  </mx:Panel>
</mx:Application>
```

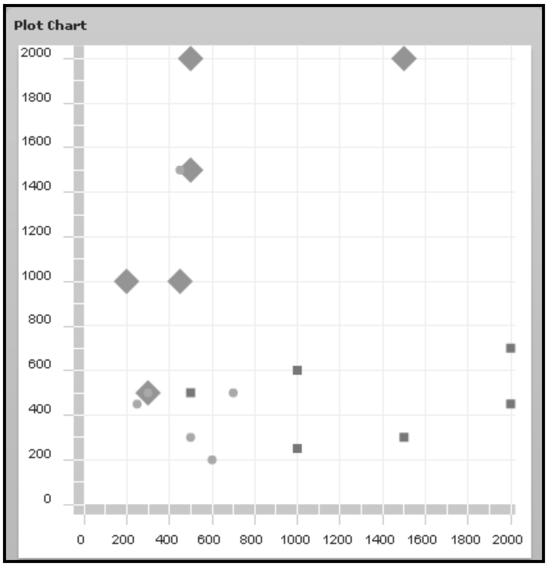
To explode one or more wedges of the pie, you use an Array of `explodeRadius` values. Each value in the Array applies to the corresponding data point. In the following example, the fourth data point, the Car expense, is exploded:

```
<?xml version="1.0"?>
<!-- charts/ExplodingPiePerWedge.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Expense:"Taxes", Amount:2000},
        {Expense:"Rent", Amount:1000},
        {Expense:"Bills", Amount:100},
        {Expense:"Car", Amount:450},
        {Expense:"Gas", Amount:100},
        {Expense:"Food", Amount:200}
     ]);

     // Create a bindable Array of explode radii.
     [Bindable]
     public var explodingArray:Array = [0,0,0,.2,0,0]
  ]]></mx:Script>
  <mx:Panel title="Exploding Pie Chart Per Wedge">
     <mx:PieChart id="pie"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:series>
           <!--Apply the Array of radii to the PieSeries.-->
           <mx:PieSeries
                field="Amount"
                nameField="Expense"
                perWedgeExplodeRadius="{explodingArray}"
                labelPosition="callout"
           />
        </mx:series>
     </mx:PieChart>
     <mx:Legend dataProvider="{pie}"/>
  </mx:Panel>
</mx:Application>
```

## Using plot charts

You use the PlotChart control to represent data in Cartesian coordinates where each data point has one value that determines its position along the x-axis, and one value that determines its position along the y-axis. You can define the shape that Flex displays at each data point with the renderer for the data series.

The following image shows an example of a plot chart:



You use the PlotSeries class with the PlotChart control to define the data for the chart. The following table describes the properties of the PlotSeries chart series that you commonly use to define your chart:

| Property | Description |
|---|---|
| `yField` | Specifies the field of the data provider that determines the y-axis location of each data point. |
| `xField` | Specifies the field of the data provider that determines the x-axis location of each data point. |
| `radius` | Specifies the radius, in pixels, of the symbol at each data point. The default value is `5` pixels. |

NOTE

Both the `xField` and `yField` properties are required for each PlotSeries in a PlotChart control.

The following example defines three data series in a PlotChart control:

```
<?xml version="1.0"?>
<!-- charts/BasicPlot.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"January", Profit:2000, Expenses:1500, Amount:450},
        {Month:"February", Profit:1000, Expenses:200, Amount:600},
        {Month:"March", Profit:1500, Expenses:500, Amount:300},
        {Month:"April", Profit:500, Expenses:300, Amount:500},
        {Month:"May", Profit:1000, Expenses:450, Amount:250},
        {Month:"June", Profit:2000, Expenses:500, Amount:700}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Plot Chart">
     <mx:PlotChart id="myChart" dataProvider="{expenses}"
     showDataTips="true">
        <mx:series>
           <mx:PlotSeries
                xField="Expenses"
                yField="Profit"
                displayName="Plot 1"
           />
           <mx:PlotSeries
                xField="Amount"
                yField="Expenses"
                displayName="Plot 2"
           />
           <mx:PlotSeries
                xField="Profit"
                yField="Amount"
                displayName="Plot 3"
           />
        </mx:series>
     </mx:PlotChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

By default, Flex displays the first data series in the chart as a diamond at each point. When you define multiple data series in a chart, Flex rotates the shape for the series (starting with a diamond, then a circle, then a square). If you have more series than there are default renderers, Flex begins again with the diamond.

The diamond shape, like the other shapes, is defined by a renderer class. The renderer classes that define these shapes are in the mx.charts.renderers package. The circle is defined by the CircleItemRenderer class. The following default renderer classes define the appearance of the data points:

- BoxItemRenderer
- CircleItemRenderer
- CrossItemRenderer
- DiamondItemRenderer
- ShadowBoxItemRenderer
- TriangleItemRenderer

You can control the image that is displayed by the chart for each data point by setting the `itemRenderer` style property of the series. The following example overrides the default renderers for the series:

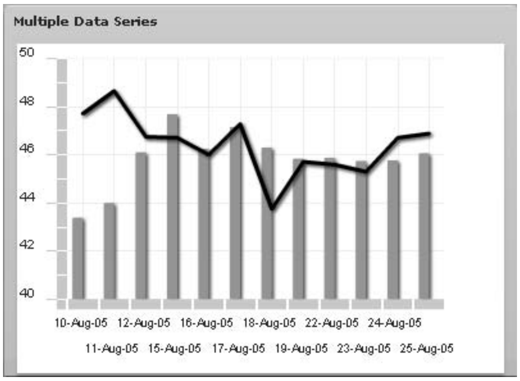
```
<?xml version="1.0"?>
<!-- charts/PlotWithCustomRenderer.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
    <![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"January", Profit:2000, Expenses:1500, Amount:450},
        {Month:"February", Profit:1000, Expenses:200, Amount:600},
        {Month:"March", Profit:1500, Expenses:500, Amount:300},
        {Month:"April", Profit:500, Expenses:300, Amount:500},
        {Month:"May", Profit:1000, Expenses:450, Amount:250},
        {Month:"June", Profit:2000, Expenses:500, Amount:700}
     ]);
    ]]>
  </mx:Script>

  <mx:Panel title="Plot Chart With Custom Item Renderer">
     <mx:PlotChart id="myChart" dataProvider="{expenses}"
     showDataTips="true">
        <mx:series>
           <mx:PlotSeries
                xField="Expenses"
                yField="Profit"
                displayName="Plot 1"
                itemRenderer="mx.charts.renderers.CrossItemRenderer"
                radius="10"
           />
           <mx:PlotSeries
                xField="Amount"
                yField="Expenses"
                displayName="Plot 2"
                itemRenderer="mx.charts.renderers.DiamondItemRenderer"
                radius="10"
           />
           <mx:PlotSeries
                xField="Profit"
                yField="Amount"
                displayName="Plot 3"
                itemRenderer=
                "mx.charts.renderers.TriangleItemRenderer"
                radius="10"
           />
        </mx:series>
    </mx:PlotChart>
    <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

You can also use graphics or custom classes to define each plot point. For more information, see "Creating custom renderers" on page 1788.

## Using multiple data series

Charts that are subclasses of the CartesianChart class let you mix different data series in the same chart control. You can create a column chart with a trend line running through it or mix any data series with any other similar series.

The following chart tracks two stocks, one represented by a column and the other represented by a line:



You can use any combination of the following series objects in a CartesianChart control:

- AreaSeries
- BarSeries
- BubbleSeries
- CandlestickSeries
- ColumnSeries
- HLOCSeries
- LineSeries
- PlotSeries

The following example mixes a LineSeries and a ColumnSeries:

```
<?xml version="1.0"?>
<!-- charts/MultipleSeries.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="500"
height="600">
  <mx:Script>
    <![CDATA[
        [Bindable]
        public var SMITH:Array = [
           {date:"22-Aug-05", close:45.87},
           {date:"23-Aug-05", close:45.74},
           {date:"24-Aug-05", close:45.77},
           {date:"25-Aug-05", close:46.06},
        ];
        [Bindable]
        public var DECKER:Array = [
           {date:"22-Aug-05", close:45.59},
           {date:"23-Aug-05", close:45.3},
           {date:"24-Aug-05", close:46.71},
           {date:"25-Aug-05", close:46.88},
        ];
    ]]>
  </mx:Script>

  <mx:Panel title="Multiple Data Series" width="400" height="400">
     <mx:ColumnChart id="mychart"
        dataProvider="{SMITH}"
        showDataTips="true"
        height="250"
        width="350"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="date"/>
        </mx:horizontalAxis>
        <mx:verticalAxis>
           <mx:LinearAxis minimum="40" maximum="50"/>
        </mx:verticalAxis>
        <mx:series>
           <mx:ColumnSeries
                dataProvider="{SMITH}"
                xField="date"
                yField="close"
                displayName="SMITH"
           >
           </mx:ColumnSeries>
           <mx:LineSeries
                dataProvider="{DECKER}"
                xField="date"
                yField="close"
                displayName="DECKER"
```

```
            >
            </mx:LineSeries>
        </mx:series>
    </mx:ColumnChart>
  </mx:Panel>
</mx:Application>
```

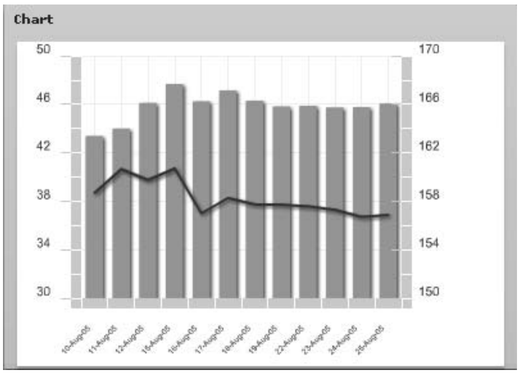
Using multiple series in the same chart works best when the data points are in a similar range (such as a stock price and its moving average). When the data points are in numerically very different ranges, the chart can be difficult to understand because the data is shown on a single axis. The solution to this problem is to use multiple axes, each with its own range. You can plot each data series on its own axis within the same chart using the techniques described in "Using multiple axes" on page 1698.

## Using multiple axes

One potential problem when using more than one data series in a single chart is that if the scales of the data are very different, the data points might be plotted in very different areas on the chart's canvas. For example, one stock price could trade in the range of $100 to $150, while another stock price could fluctuate from $2 to $2.50. If you plot both stocks in the same chart, it would be difficult to see any correlation between the prices, even with a logarithmic axis.

To work around this problem, you use multiple axes in your charts so that each data series is positioned relative to its own axis. All chart controls that are subclasses of CartesianChart support adding additional sets of data on a additional scales in the horizontal axis, vertical axis, or both. (This applies to all charts except the PieChart control.) You can use values on the additional axes to compare multiple sets of data that are on different scales, such as stock prices that trade in different ranges.

The following example shows a stock price that trades within a $40 to 45 range, and another stock price that trades within a $150 to 160 range. The values of the axis on the left show the range of values of the first stock, and the values of the axis on the right show the range of values of the second stock:



To use multiple axes in a chart, you first define the chart's series and their axes. For example, for a chart that mixes with columns with a line, you would have a ColumnSeries and LineSeries. For each of these, you would likely also define the vertical axis, as the following example shows:

```
<mx:series>
  <mx:ColumnSeries id="cs1" dataProvider="{SMITH}" yField="close">
    <mx:verticalAxis>
      <mx:LinearAxis id="v1" minimum="40" maximum="50"/>
    </mx:verticalAxis>
  </mx:ColumnSeries>
  <mx:LineSeries id="cs2" dataProvider="{DECKER}" yField="close">
    <mx:verticalAxis>
      <mx:LinearAxis id="v2" minimum="150" maximum="170"/>
    </mx:verticalAxis>
  </mx:LineSeries>
</mx:series>
```

You then define the axis renderers, and bind their `axis` properties to the series' axes. In this case, you define two vertical axis renderers, and bind them to the LinearAxis objects:

```
<mx:verticalAxisRenderers>
  <mx:AxisRenderer placement="left" axis="{v1}"/>
  <mx:AxisRenderer placement="left" axis="{v2}"/>
</mx:verticalAxisRenderers>
```

Note that you control the location of the axis by using the `placement` property of the AxisRenderer. For vertical axis renderers, valid values are `left` and `right`. For horizontal axis renderers, valid values are `top` and `bottom`.

Axes can be independent of the series definition, too. For example, you can also point more than one series' to the same axis. In this case, you could define a horizontal axis:

```
<mx:horizontalAxis>
  <mx:CategoryAxis id="h1" categoryField="date"/>
<mx:horizontalAxis>
```

And then bind it to the series:

```
<mx:ColumnSeries id="cs1" horizontalAxis="{h1}" dataProvider="{SMITH}"
  yField="close">
...
<mx:LineSeries id="cs2" horizontalAxis="{h1}" dataProvider="{DECKER}"
  yField="close">
```

And you can bind an axis renderer to that same axis:

```
<mx:horizontalAxisRenderers>
  <mx:AxisRenderer placement="bottom" axis="{h1}"/>
</mx:horizontalAxisRenderers>
```

The final result is a chart with multiple axes, but whose series share some of the same properties defined by common axis renderers:

```
<?xml version="1.0"?>
<!-- charts/MultipleAxes.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
      public var SMITH:ArrayCollection = new ArrayCollection([
        {date:"22-Aug-05", close:41.87},
        {date:"23-Aug-05", close:45.74},
        {date:"24-Aug-05", close:42.77},
        {date:"25-Aug-05", close:48.06},
     ]);

     [Bindable]
      public var DECKER:ArrayCollection = new ArrayCollection([
        {date:"22-Aug-05", close:157.59},
        {date:"23-Aug-05", close:160.3},
        {date:"24-Aug-05", close:150.71},
        {date:"25-Aug-05", close:156.88},
     ]);

  ]]></mx:Script>

  <mx:Panel title="Column Chart With Multiple Axes">
     <mx:ColumnChart id="myChart" showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis id="h1" categoryField="date"/>
        </mx:horizontalAxis>

        <mx:horizontalAxisRenderers>
            <mx:AxisRenderer placement="bottom" axis="{h1}"/>
        </mx:horizontalAxisRenderers>

        <mx:verticalAxisRenderers>
            <mx:AxisRenderer placement="left" axis="{v1}"/>
            <mx:AxisRenderer placement="left" axis="{v2}"/>
        </mx:verticalAxisRenderers>

        <mx:series>
           <mx:ColumnSeries id="cs1"
                horizontalAxis="{h1}"
                dataProvider="{SMITH}"
                yField="close"
                displayName="SMITH"
            >
                <mx:verticalAxis>
```

```
                    <mx:LinearAxis id="v1" minimum="40" maximum="50"/>
                </mx:verticalAxis>
            </mx:ColumnSeries>
            <mx:LineSeries id="cs2"
                horizontalAxis="{h1}"
                dataProvider="{DECKER}"
                yField="close"
                displayName="DECKER"
            >
                <mx:verticalAxis>
                <mx:LinearAxis id="v2" minimum="150" maximum="170"/>
                </mx:verticalAxis>
            </mx:LineSeries>
        </mx:series>
     </mx:ColumnChart>
  </mx:Panel>
</mx:Application>
```

Even if the verticalAxisRenderers or horizontalAxisRenderers have not been specified, cartesian charts will create default horizontal and vertical axis renderers based on the default axes of the chart.

When using multiple axes, it is important to recognize that it will not necessarily be immediately apparent which axis applies to which data set in the chart. As a result, you should try to style the axes so that they match the styles of the chart items.

The following example defines two colors and then uses those colors in the axis renderers and in the strokes and fills for the chart items.

```
<?xml version="1.0"?>
<!-- charts/StyledMultipleAxes.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
      public var SMITH:ArrayCollection = new ArrayCollection([
        {date:"22-Aug-05", close:41.87},
        {date:"23-Aug-05", close:45.74},
        {date:"24-Aug-05", close:42.77},
        {date:"25-Aug-05", close:48.06},
     ]);

     [Bindable]
      public var DECKER:ArrayCollection = new ArrayCollection([
        {date:"22-Aug-05", close:157.59},
        {date:"23-Aug-05", close:160.3},
        {date:"24-Aug-05", close:150.71},
        {date:"25-Aug-05", close:156.88},
     ]);

    [Bindable]
    public var deckerColor:Number = 0x224488;

    [Bindable]
    public var smithColor:Number = 0x884422;
  ]]></mx:Script>

  <mx:Stroke id="h1Stroke"
        color="{smithColor}"
        weight="8"
        alpha=".75"
        caps="square"
  />

  <mx:Stroke id="h2Stroke"
        color="{deckerColor}"
        weight="8"
        alpha=".75"
        caps="square"
  />

  <mx:Panel title="Column Chart With Multiple Axes">
     <mx:ColumnChart id="myChart" showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis id="h1" categoryField="date"/>
        </mx:horizontalAxis>

        <mx:horizontalAxisRenderers>
            <mx:AxisRenderer placement="bottom" axis="{h1}"/>
        </mx:horizontalAxisRenderers>

        <mx:verticalAxisRenderers>
            <mx:AxisRenderer placement="left" axis="{v1}">
                <mx:axisStroke>{h1Stroke}</mx:axisStroke>
            </mx:AxisRenderer>
            <mx:AxisRenderer placement="left" axis="{v2}">
                <mx:axisStroke>{h2Stroke}</mx:axisStroke>
            </mx:AxisRenderer>
        </mx:verticalAxisRenderers>

        <mx:series>
           <mx:ColumnSeries id="cs1"
                horizontalAxis="{h1}"
                dataProvider="{SMITH}"
                yField="close"
                displayName="SMITH"
            >
                <mx:fill>
                    <mx:SolidColor color="{smithColor}"/>
                </mx:fill>

                <mx:verticalAxis>
                   <mx:LinearAxis id="v1" minimum="40" maximum="50"/>
                </mx:verticalAxis>
           </mx:ColumnSeries>
           <mx:LineSeries id="cs2"
                horizontalAxis="{h1}"
                dataProvider="{DECKER}"
                yField="close"
                displayName="DECKER"
            >
                <mx:verticalAxis>
               <mx:LinearAxis id="v2" minimum="150" maximum="170"/>
                </mx:verticalAxis>

                <mx:lineStroke>
                    <mx:Stroke
                        color="{deckerColor}"
                        weight="4"
                        alpha="1"
                    />
                </mx:lineStroke>
           </mx:LineSeries>
        </mx:series>
  </mx:ColumnChart>
```

```
    <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

You can customize labels by using the `labelFunction` property of the AxisRenderer class. This lets you control the labels if you use multiple axes. For more information, see "Customizing axis labels" on page 1824.

For CartesianChart controls, there is no limit to the number of axes you can have.

For PolarChart controls, such as a PieChart, you generally do not use multiple axes because even though each series could have its own angular axis, the angular axis is always from 0 to 360 (the size of the wedge).

CHAPTER 48

# Formatting Charts

48

You can customize the look and feel of your Adobe Flex Charting components. You can format the appearance of almost all chart elements, from the font properties of an axis label to the width of the stroke in a legend.

To set style properties in your chart controls, you can use Cascading Style Sheets (CSS) syntax or set the style properties inline as tag attributes. As with all styles, you can call the `setStyle()` method to set any style on your chart elements. For more information on using the `setStyle()` method, see "Using the setStyle() and getStyle() methods" on page 808.

## Contents

## Applying chart styles

You can apply style properties to charts by using CSS or inline syntax. You can also apply styles to chart elements by using binding.

### Applying styles with CSS

You can apply styles to Flex Charting components with CSS definitions. You can set chart properties such as fonts and tick marks, or series properties, such as the fills of the boxes in a ColumnChart.

A limitation of using CSS to style your charts is that the styleable chart properties often use compound values, such as strokes and gradient fills, that cannot be expressed by using CSS. The result is that you cannot express all values of chart styles by using CSS syntax.

#### Applying CSS to chart controls

You can use the control name in CSS to define styles for that control. This is referred to as a type selector, because the style you define is applied to all controls of that type. For example, the following style definition specifies the font for all BubbleChart controls:

```
<?xml version="1.0"?>
<!-- charts/BubbleStyles.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Style>
      BubbleChart {
         fontFamily:Arial;
         fontSize:20;
         color:#FF0033;
      }
    </mx:Style>
    <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:120, Amount:45},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:60},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:30}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Bubble Chart">
     <mx:BubbleChart id="myChart"
        maxRadius="50"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:series>
```

```
            <mx:BubbleSeries
                 xField="Profit"
                 yField="Expenses"
                 radiusField="Amount"
            />
         </mx:series>
      </mx:BubbleChart>
   </mx:Panel>
</mx:Application>
```

Some styles, such as `fontSize` and `fontFamily`, are inheritable, which means that if you set them on the chart control, the axes labels, titles, and other text elements on the chart inherit those settings. To determine whether a style is inheritable, see that style's description in *Adobe Flex Language Reference*.

Axis labels appear next to the tick marks along the chart's axis. Titles appear parallel to the axis line. By default, the text on an axis uses the text styles of the chart control.

Axis elements use whatever styles are set on the chart control's type or class selector, but you can also specify different styles for each axis by using a class selector on the axis renderer or predefined axis style properties.

## Applying different styles to each series

To apply different styles to each series in the charts with CSS, you use the `chartSeriesStyles` property. This property takes an Array of Strings. Each String specifies the name of a class selector in the style sheet. Flex applies each of these class selectors to a series.

To apply CSS to a series, you define a type or class selector for the chart that defines the `chartSeriesStyles` property. You then define each class selector named in the `chartSeriesStyles` property.

Essentially, you are defining a new style for each series in your chart. For example, if you have a ColumnChart control with two series, you can apply a different style to each series without having to explicitly set the styles on each series.

The following example defines the colors for two series in the ColumnChart control:

```
<?xml version="1.0"?>
<!-- charts/SeriesStyles.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Style>
     ColumnChart {
        chartSeriesStyles: PCCSeries1, PCCSeries2;
     }
     .PCCSeries1 {
        fill: #CCFF66;
```

```
      }
      .PCCSeries2 {
         fill: #CCFF99;
      }
   </mx:Style>

   <mx:Script>
   import mx.collections.ArrayCollection;
   [Bindable]
   public var expenses:ArrayCollection = new ArrayCollection([
      {Month: "Jan", Profit: 2000, Expenses: 1500},
      {Month: "Feb", Profit: 1000, Expenses: 200},
      {Month: "Mar", Profit: 1500, Expenses: 500}
   ]);
   </mx:Script>

   <mx:Panel>
      <mx:ColumnChart id="column" dataProvider="{expenses}">
         <mx:horizontalAxis>
            <mx:CategoryAxis
                 dataProvider="{expenses}"
                 categoryField="Month"
            />
         </mx:horizontalAxis>
         <mx:series>
            <mx:ColumnSeries
                 xField="Month"
                 yField="Profit"
                 displayName="Profit"
            />
            <mx:ColumnSeries
                 xField="Month"
                 yField="Expenses"
                 displayName="Expenses"
            />
         </mx:series>
      </mx:ColumnChart>
      <mx:Legend dataProvider="{column}"/>
   </mx:Panel>
</mx:Application>
```

Flex sets the `styleName` property of each series to the corresponding selector in the `chartSeriesStyles` Array. You do not have to explicitly set styles for each series. If you have more series than available `chartSeriesStyles` selectors, the chart begins again with the first style.

If you manually set the value of the `styleName` property on a particular series, that style takes priority over the styles that the `chartSeriesStyles` property specifies.

For PieChart controls, you can define the `fills` property of the series. The PieChart applies each of the values in this Array for each pie wedge. It starts with the first value if there are more wedges than values defined in the Array. The following example creates a PieChart that uses only red, white, and blue colors for the wedges:

```
<?xml version="1.0"?>
<!-- charts/PieWedgeFills.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
      PieSeries {
         fills:#FF0000, #FFFFFF, #006699;
      }
  </mx:Style>

  <mx:Script>
    <![CDATA[
        import mx.collections.ArrayCollection;
        [Bindable]
        public var expenses:ArrayCollection = new ArrayCollection([
           {Expense:"Taxes", Amount:2000},
           {Expense:"Rent", Amount:1000},
           {Expense:"Bills", Amount:100},
           {Expense:"Car", Amount:450},
           {Expense:"Gas", Amount:100},
           {Expense:"Food", Amount:200}
        ]);
    ]]>
  </mx:Script>
  <mx:Panel>
     <mx:PieChart id="pie"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:series>
           <mx:PieSeries
                field="Amount"
                nameField="Expense"
                labelPosition="callout"
           />
        </mx:series>
     </mx:PieChart>
     <mx:Legend dataProvider="{pie}"/>
  </mx:Panel>
</mx:Application>
```

## Using predefined axis style properties

The following predefined class selectors are available for axis styles:

- `horizontalAxisStyleName`
- `verticalAxisStyleName`

Flex applies each style to the corresponding axis.

In addition to these, there are also two class selectors that define an array of class selectors that define the style properties for the axes:

- `horizontalAxisStyleNames`
- `verticalAxisStyleNames`

In this case, Flex loops through the selectors and applies them to the axis renderers that correspond to their position in the Array.

In addition to these class selectors for the axis style properties, you can use the `axisTitleStyleName` and `gridLinesStyleName` class selectors to apply styles to axis titles and grid lines.

The following example removes tick marks from the horizontal axis:

```
<?xml version="1.0"?>
<!-- charts/PredefinedAxisStyles.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     ColumnChart {
        horizontalAxisStyleName:myAxisStyles;
        verticalAxisStyleName:myAxisStyles;
     }

     .myAxisStyles {
        tickPlacement:none;
     }
  </mx:Style>

  <mx:Script>
    <![CDATA[
        import mx.collections.ArrayCollection;
        [Bindable]
        public var expenses:ArrayCollection = new ArrayCollection([
           {Month:"Jan", Profit:2000, Expenses:1500},
           {Month:"Feb", Profit:1000, Expenses:200},
           {Month:"Mar", Profit:1500, Expenses:500}
        ]);
    ]]>
  </mx:Script>

  <mx:Panel>
     <mx:ColumnChart id="column" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
               displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
```

```
            />
         </mx:series>
      </mx:ColumnChart>
      <mx:Legend dataProvider="{column}"/>
   </mx:Panel>
</mx:Application>
```

## Using class selectors for axis styles

To use a class selector to define styles for axis elements, define the custom class selector in an `<mx:Style>` block or external style sheet, and then use the `styleName` property of the AxisRenderer class to point to that class selector.

The following example defines the `MyStyle` style and applies that style to the elements on the horizontal axis:

```
<?xml version="1.0"?>
<!-- charts/AxisClassSelectors.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

<mx:Style>
   .myStyle {
      fontSize:7;
      color:red;
   }
</mx:Style>

   <mx:Script><![CDATA[
   import mx.collections.ArrayCollection;
   [Bindable]
   public var expenses:ArrayCollection = new ArrayCollection([
      {Month:"Jan",Profit:2000,Expenses:1500},
      {Month:"Feb",Profit:1000,Expenses:200},
      {Month:"Mar",Profit:1500,Expenses:500}
   ]);
   ]]></mx:Script>

   <mx:Panel>
      <mx:ColumnChart id="column" dataProvider="{expenses}">

         <mx:horizontalAxisRenderers>
            <mx:AxisRenderer styleName="myStyle"/>
         </mx:horizontalAxisRenderers>

         <mx:horizontalAxis>
            <mx:CategoryAxis
                 dataProvider="{expenses}"
                 categoryField="Month"
            />
         </mx:horizontalAxis>
```

```
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{column}"/>
  </mx:Panel>
</mx:Application>
```

Most charting-specific style properties are not inheritable, which means that if you set the property on a parent object, the child object does not inherit its value.

For more information on using CSS, see "About styles" on page 767.

## Applying styles inline

You can set many styleable elements of a chart as attributes of the MXML tag. For example, to set the styleable properties of an axis, you can use the `<mx:AxisRenderer>` tag rather than define a new style in CSS.

The following example sets the `fontSize` property of the horizontal axis to 7:

```
<mx:horizontalAxisRenderers>
  <mx:AxisRenderer fontSize="7"/>
</mx:horizontalAxisRenderers>
```

You can also access the properties of renderers in ActionScript so that you can change their appearance at run time. For additional information about axis renderers, see "Working with axes" on page 1795.

## Applying styles by binding tag definitions

You can define styles with MXML tags. You can then bind the values of the renderer properties to those tags. The following example defines the weight and color of the strokes, and then applies those strokes to the chart's AxisRenderer class:

```
<?xml version="1.0"?>
<!-- charts/BindStyleValues.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
  [Bindable]
  public var aapl:Array = [
     {date:"1-Aug-05",open:42.57,high:43.08,low:42.08,close:
     42.75},
     {date:"2-Aug-05",open:42.89,high:43.5,low:42.61,close:
     43.19},
     {date:"3-Aug-05",open:43.19,high:43.31,low:42.77,close:
     43.22},
     {date:"4-Aug-05",open:42.89,high:43,low:42.29,close:42.71},
     {date:"5-Aug-05",open:42.49,high:43.36,low:42.02,close:
     42.99},
     {date:"8-Aug-05",open:43,high:43.25,low:42.61,close:42.65},
     {date:"9-Aug-05",open:42.93,high:43.89,low:42.91,close:
     43.82},
     {date:"10-Aug-05",open:44,high:44.39,low:43.31,close:43.38},
     {date:"11-Aug-05",open:43.39,high:44.12,low:43.25,close:44},
     {date:"12-Aug-05",open:43.46,high:46.22,low:43.36,close:
     46.1},
  ];
  ]]></mx:Script>

  <mx:Stroke color="0x00FF00" weight="2" id="axis"/>
  <mx:Stroke color="0xFF0000" weight="1" id="ticks"/>
  <mx:Stroke color="0x0000FF" weight="1" id="mticks"/>

  <mx:HLOCChart id="mychart"
    dataProvider="{aapl}"
    showDataTips="true"
  >
     <mx:horizontalAxisRenderers>
        <mx:AxisRenderer
            axisStroke="{axis}"
            placement="bottom"
            minorTickPlacement="inside"
            minorTickLength="2"
            tickLength="5"
            tickPlacement="inside"
        >
           <mx:tickStroke>{ticks}</mx:tickStroke>
           <mx:minorTickStroke>{mticks}</mx:minorTickStroke>
        </mx:AxisRenderer>
     </mx:horizontalAxisRenderers>

     <mx:verticalAxis>
        <mx:LinearAxis minimum="30" maximum="50"/>
     </mx:verticalAxis>
     <mx:series>
        <mx:HLOCSeries
            dataProvider="{aapl}"
            openField="open"
```

```
            highField="high"
            lowField="low"
            closeField="close"
            displayName="AAPL"
        >
        </mx:HLOCSeries>
     </mx:series>
  </mx:HLOCChart>
  <mx:Legend dataProvider="{mychart}"/>
</mx:Application>
```

## Using ChartElement objects

The ChartElement class is the base class for anything that appears in the data area of the chart. All series objects (such as GridLines objects) are ChartElement objects. You can add ChartElement objects (such as images, grid lines, and strokes) to your charts by using the `backgroundElements` and `annotationElements` properties of the chart classes.

The `backgroundElements` property specifies an Array of ChartElement objects that appear beneath any data series rendered by the chart. The `annotationElements` property specifies an Array of ChartElement objects that appears above any data series rendered by the chart.

The ChartElement objects that you can add to a chart include supported image files, such as GIF, SVG, and JPEG.

The following example adds new grid lines as annotation elements to the chart and an image as the background element. When the user clicks the button, the annotation elements change:

```
<?xml version="1.0"?>
<!-- charts/AnnotationElements.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
  import mx.collections.ArrayCollection;
  import mx.graphics.SolidColor;
  import mx.charts.GridLines;

  [Bindable]
  public var expenses:ArrayCollection = new ArrayCollection([
     {Month:"Jan", Profit:2000, Expenses:1500},
     {Month:"Feb", Profit:1000, Expenses:200},
     {Month:"Mar", Profit:1500, Expenses:500}
  ]);

  [Embed(source="../assets/bird.gif")]
  public var bird:Class;

  public function updateGridLines():void {
     var bgi:GridLines = new GridLines();

     var s:Stroke = new Stroke(0xff00ff, 3);
     bgi.setStyle("horizontalStroke",s);

     var c:SolidColor = new SolidColor(0x990033, .2);
     bgi.setStyle("horizontalFill",c);

     var c2:SolidColor = new SolidColor(0x999933, .2);
     bgi.setStyle("horizontalAlternateFill",c2);

     myChart.annotationElements = [bgi]

     var b:Object = new bird();
     b.alpha = .2;
     b.height = 150;
     b.width = 150;

     myChart.backgroundElements = [ b ];
  }
  ]]></mx:Script>
  <mx:Panel>
     <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>

        <mx:annotationElements>
           <mx:GridLines>
            <mx:horizontalStroke>
                <mx:Stroke
                    color="#191970"
                    weight="2"
                    alpha=".3"
                />
            </mx:horizontalStroke>
           </mx:GridLines>
        </mx:annotationElements>

        <mx:backgroundElements>
           <mx:Image
                source="@Embed('../assets/bird.gif')"
                alpha=".2"
           />
        </mx:backgroundElements>

     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
     <mx:Button id="b1"
```

```
        click="updateGridLines()"
        label="Update Grid Lines"
     />
  </mx:Panel>
</mx:Application>
```

In ActionScript, you can add an image to the chart and manipulate its properties, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/BackgroundElementsWithActionScript.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="addBird()">
  <mx:Script><![CDATA[
  import mx.collections.ArrayCollection;
  import mx.graphics.SolidColor;
  import mx.charts.GridLines;

  [Bindable]
  public var expenses:ArrayCollection = new ArrayCollection([
     {Month:"Jan", Profit:2000, Expenses:1500},
     {Month:"Feb", Profit:1000, Expenses:200},
     {Month:"Mar", Profit:1500, Expenses:500}
  ]);

  [Embed(source="../assets/bird.gif")]
  public var bird:Class;

  public function addBird():void {
     var b:Object = new bird();
     b.alpha = .2;
     myChart.backgroundElements = [ b ];
  }
  ]]></mx:Script>
  <mx:Panel>
     <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
            />
        </mx:series>
    </mx:ColumnChart>
    <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

For more information on working with grid lines, see “Using grid lines” on page 1768.

You can also add data graphics to your charts by adding the CartesianDataCanvas and PolarDataCanvas controls to your background or annotation elements Arrays. For more information, see “Drawing on chart controls” on page 1966.

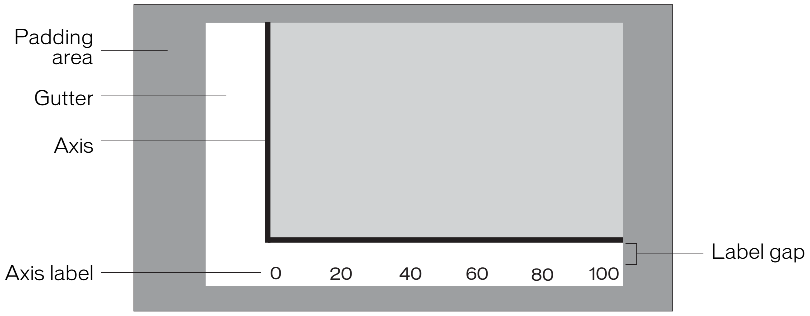
## Setting padding properties

As with other Flex components, the padding properties of a chart define an area between the outside bounds of the chart control and its content. Flex draws only the background fill in the padding area.

You can set the padding values for a chart control by using the `paddingLeft` and `paddingRight` properties that the chart control inherits from the UIComponent class. You can also use the `paddingTop` and `paddingBottom` properties that the chart control inherits from the ChartBase class.

Flex Charting elements also have gutters. The *gutter* is the area between the padding area and the actual axis line. Flex draws labels, titles, and tick marks for the axes in the gutter of the chart. Chart controls adjust the gutters to accommodate these enhancements to the axis, but you can specify explicit gutter values.

The following example shows the locations of the gutters and the padding area on a chart:



The following style properties define the size of a chart's gutters:

- `gutterLeft`
- `gutterRight`
- `gutterTop`
- `gutterBottom`

The default value of the gutter styles is `undefined`, which means that the chart determines appropriate values. Overriding the default value and explicitly setting gutter values can improve the speed of rendering charts, because Flex does not have to dynamically calculate the gutter size. However, it can also cause clipping of axis labels or other undesirable effects.

You set gutter styles on the chart control. The following example creates a region that is 50 pixels wide for the axis labels, titles, and tick marks, by explicitly setting the values of the `gutterLeft`, `gutterRight`, and `gutterBottom` style properties. It also sets the `paddingTop` property to 20.

```
<?xml version="1.0"?>
<!-- charts/GutterStyles.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Style>
     ColumnChart {
        gutterLeft:50;
        gutterRight:50;
        gutterBottom:50;
        paddingTop:20;
     }
  </mx:Style>

  <mx:Script><![CDATA[
  import mx.collections.ArrayCollection;
  [Bindable]
  public var expenses:ArrayCollection = new ArrayCollection([
     {Month:"Jan", Profit:2000, Expenses:1500},
     {Month:"Feb", Profit:1000, Expenses:200},
     {Month:"Mar", Profit:1500, Expenses:500}
  ]);
  ]]></mx:Script>
  <mx:Panel>
     <mx:ColumnChart id="column" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
            />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{column}"/>
  </mx:Panel>
</mx:Application>
```

Alternatively, you can set the gutter properties inline, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/GutterProperties.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
  import mx.collections.ArrayCollection;
  [Bindable]
  public var expenses:ArrayCollection = new ArrayCollection([
     {Month:"Jan", Profit:2000, Expenses:1500},
     {Month:"Feb", Profit:1000, Expenses:200},
     {Month:"Mar", Profit:1500, Expenses:500}
  ]);
  ]]></mx:Script>
  <mx:Panel>
     <mx:ColumnChart id="myChart"
        dataProvider="{expenses}"
        gutterLeft="50"
        gutterRight="50"
        gutterBottom="50"
        gutterTop="20"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

In addition to the gutter and padding properties, you can set the `labelGap` property of a chart's axes. The `labelGap` property defines the distance, in pixels, between the tick marks and the axis labels. You set the `labelGap` property on the AxisRenderer tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/LabelGaps.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
      import mx.collections.ArrayCollection;
      [Bindable]
      public var expenses:ArrayCollection = new ArrayCollection([
         {Month:"Jan", Profit:2000, Expenses:1500},
         {Month:"Feb", Profit:1000, Expenses:200},
         {Month:"Mar", Profit:1500, Expenses:500}
      ]);
  ]]></mx:Script>
  <mx:Panel>
     <mx:ColumnChart id="myChart"
        dataProvider="{expenses}"
        gutterLeft="50"
        gutterRight="50"
        gutterBottom="50"
        gutterTop="20"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:horizontalAxisRenderers>
           <mx:AxisRenderer labelGap="20"/>
        </mx:horizontalAxisRenderers>
        <mx:verticalAxisRenderers>
           <mx:AxisRenderer labelGap="20"/>
        </mx:verticalAxisRenderers>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
```

```
    <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

## Formatting tick marks

There are two types of tick marks on a Flex chart: major and minor. Major tick marks are the indications along an axis that correspond to an axis label. The text for the axis labels is often derived from the chart's data provider. Minor tick marks are those tick marks that appear between the major tick marks. Minor tick marks help the user visualize the distance between the major tick marks.

You use the `tickPlacement` and `minorTickPlacement` properties of the AxisRenderer object to determine whether or not Flex displays tick marks and where Flex displays tick marks.

The following table describes valid values of the `tickPlacement` and `minorTickPlacement` properties:

| Value | Description |
|---|---|
| `cross` | Places tick marks across the axis. |
| `inside` | Places tick marks on the inside of the axis line. |
| `none` | Hides tick marks. |
| `outside` | Places tick marks on the outside of the axis line. |

You can align the tick marks with labels by using the `tickAlignment` property.

Flex also lets you set the length of tick marks and the number of minor tick marks that appear along the axis. The following table describes the properties that define the length of tick marks on the chart's axes:

| Property | Description |
|---|---|
| `tickLength` | The length, in pixels, of the major tick mark from the axis. |
| `minorTickLength` | The length, in pixels, of the minor tick mark from the axis. |

The minor tick marks overlap the placement of major tick marks. So, if you hide major tick marks but still show minor tick marks, the minor tick marks appear at the regular tick-mark intervals.

The following example sets tick marks to the inside of the axis line, sets the tick mark's length to 12 pixels, and hides minor tick marks:

```
<?xml version="1.0"?>
<!-- charts/TickStyles.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
  [Bindable]
  public var aapl:Array = [
     {date:"1-Aug-05",open:42.57,high:43.08,low:42.08,close:42.75},
     {date:"2-Aug-05",open:42.89,high:43.5,low:42.61,close:43.19},
     {date:"3-Aug-05",open:43.19,high:43.31,low:42.77,
     close:43.22},
     {date:"4-Aug-05",open:42.89,high:43,low:42.29,close:42.71},
     {date:"5-Aug-05",open:42.49,high:43.36,low:42.02,
     close:42.99},
     {date:"8-Aug-05",open:43,high:43.25,low:42.61,close:42.65},
     {date:"9-Aug-05",open:42.93,high:43.89,low:42.91,
     close:43.82},
     {date:"10-Aug-05",open:44,high:44.39,low:43.31,close:43.38},
     {date:"11-Aug-05",open:43.39,high:44.12,low:43.25,close:44},
     {date:"12-Aug-05",open:43.46,high:46.22,low:43.36,
     close:46.1},
  ];
  ]]></mx:Script>

  <mx:Style>
     .myAxisStyle {
        placement:bottom;
        minorTickPlacement:none;
        tickLength:12;
        tickPlacement:inside;
     }
  </mx:Style>

   <mx:HLOCChart id="mychart"
```

```
        dataProvider="{aapl}"
        showDataTips="true"
    >
    <mx:horizontalAxisRenderers>
        <mx:AxisRenderer styleName="myAxisStyle"/>
    </mx:horizontalAxisRenderers>

    <mx:verticalAxis>
        <mx:LinearAxis minimum="30" maximum="50"/>
    </mx:verticalAxis>

    <mx:series>
        <mx:HLOCSeries
            dataProvider="{aapl}"
            openField="open"
            highField="high"
            lowField="low"
            closeField="close"
            displayName="AAPL"
        />
     </mx:series>
  </mx:HLOCChart>
  <mx:Legend dataProvider="{mychart}"/>
</mx:Application>
```

## Formatting axis lines

Axes have lines to which the tick marks are attached. You can use style properties to hide these lines or change the width of the lines.

To hide the axis line, set the value of the `showLine` property on the AxisRenderer object to `false`. The default value is `true`. The following example sets `showLine` to `false`:

```
<?xml version="1.0"?>
<!-- charts/DisableAxisLines.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Line Chart">
     <mx:LineChart dataProvider="{expenses}" showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:horizontalAxisRenderers>
           <mx:AxisRenderer showLine="false"/>
        </mx:horizontalAxisRenderers>
        <mx:verticalAxisRenderers>
           <mx:AxisRenderer showLine="false"/>
        </mx:verticalAxisRenderers>
        <mx:series>
           <mx:LineSeries
                yField="Profit"
                displayName="Profit"
           />
           <mx:LineSeries
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:LineChart>
  </mx:Panel>
</mx:Application>
```

You can also apply the `showLine` property as a CSS style property.

You can change the `width`, `color`, and `alpha` of the axis line with the `<mx:axisStroke>` tag. You use an `<mx:Stroke>` child tag to define these properties or define a stroke and then bind it to the `axisStroke` object, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/StyleAxisLines.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);
  ]]></mx:Script>

  <mx:Stroke id="axisStroke"
        color="#884422"
        weight="8"
        alpha=".75"
        caps="square"
  />

  <mx:Panel title="Line Chart">
     <mx:LineChart dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
            />
        </mx:horizontalAxis>
        <mx:horizontalAxisRenderers>
           <mx:AxisRenderer>
            <mx:axisStroke>{axisStroke}</mx:axisStroke>
           </mx:AxisRenderer>
        </mx:horizontalAxisRenderers>
        <mx:verticalAxisRenderers>
           <mx:AxisRenderer>
            <mx:axisStroke>
                <mx:Stroke color="#884422"
                    weight="8"
                    alpha=".75"
                    caps="square"
                />
            </mx:axisStroke>
           </mx:AxisRenderer>
        </mx:verticalAxisRenderers>
        <mx:series>
           <mx:LineSeries
                yField="Profit"
                displayName="Profit"
           />
           <mx:LineSeries
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:LineChart>
  </mx:Panel>
</mx:Application>
```

For more information about strokes, see “Using strokes” on page 1731.

You can change the placement of the axis lines (for example, move the axis line from the bottom of the chart control to the top) by using the `placement` property of the AxisRenderer. For more information, see “Positioning the axes” on page 1777.

You can also apply filters to axis lines to further customize their appearance. For more information, see “Using filters with chart controls” on page 1759.

## Using strokes

You use the Stroke class with the chart series and grid lines to control the properties of the lines that Flex uses to draw chart elements.

The following table describes the properties that you use to control the appearance of strokes:

| Property | Description |
|---|---|
| `color` | Specifies the color of the line as a hexadecimal value. The default value is `0x000000`, which corresponds to black. |
| `weight` | Specifies the width of the line, in pixels. The default value is 0, which corresponds to a hairline. |
| `alpha` | Specifies the transparency of a line. Valid values are 0 (invisible) through 100 (opaque). The default value is 100. |

The following example defines a line width of 2 pixels, one with a dark gray border (`0x808080`) and the other with a light gray border (`0xC0C0C0`) for the borders of chart items in a BarChart control's BarSeries:

```
<?xml version="1.0"?>
<!-- charts/BasicBarStroke.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Bar Chart">
     <mx:BarChart id="myChart" dataProvider="{expenses}">
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries
                yField="Month"
                xField="Profit"
                displayName="Profit"
            >
            <mx:stroke>
                <mx:Stroke
                    color="0x808080"
                    weight="2"
                    alpha=".8"
                />
            </mx:stroke>
           </mx:BarSeries>
           <mx:BarSeries
                yField="Month"
                xField="Expenses"
                displayName="Expenses"
           >
            <mx:stroke>
                <mx:Stroke
                    color="0xC0C0C0"
                    weight="2"
                    alpha=".8"
                />
            </mx:stroke>
           </mx:BarSeries>
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

## Defining AxisRenderer properties with strokes

You can use strokes to define tick marks and other properties of an AxisRenderer, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/AxisRendererStrokes.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
  [Bindable]
  public var aapl:Array = [
     {date:"1-Aug-05",open:42.57,high:43.08,low:42.08,close:42.75},
     {date:"2-Aug-05",open:42.89,high:43.5,low:42.61,close:43.19},
     {date:"3-Aug-05",open:43.19,high:43.31,low:42.77,close:43.22},
     {date:"4-Aug-05",open:42.89,high:43,low:42.29,close:42.71},
     {date:"5-Aug-05",open:42.49,high:43.36,low:42.02,close:42.99},
     {date:"8-Aug-05",open:43,high:43.25,low:42.61,close:42.65},
     {date:"9-Aug-05",open:42.93,high:43.89,low:42.91,close:43.82},
     {date:"10-Aug-05",open:44,high:44.39,low:43.31,close:43.38},
     {date:"11-Aug-05",open:43.39,high:44.12,low:43.25,close:44},
     {date:"12-Aug-05",open:43.46,high:46.22,low:43.36,close:46.1},
  ];
  ]]></mx:Script>

  <mx:HLOCChart id="myChart"
    dataProvider="{aapl}"
    showDataTips="true"
  >
     <mx:horizontalAxisRenderers>
        <mx:AxisRenderer
            placement="bottom"
            canDropLabels="true"
            tickPlacement="inside"
            tickLength="10"
            minorTickPlacement="inside"
            minorTickLength="5"
        >
           <mx:axisStroke>
            <mx:Stroke color="#000080" weight="1"/>
           </mx:axisStroke>
           <mx:tickStroke>
            <mx:Stroke color="#000060" weight="1"/>
           </mx:tickStroke>
           <mx:minorTickStroke>
            <mx:Stroke color="#100040" weight="1"/>
           </mx:minorTickStroke>
        </mx:AxisRenderer>
     </mx:horizontalAxisRenderers>

     <mx:verticalAxis>
```

```
         <mx:LinearAxis minimum="30" maximum="50"/>
      </mx:verticalAxis>

      <mx:series>
         <mx:HLOCSeries
             dataProvider="{aapl}"
             openField="open"
             highField="high"
             lowField="low"
             closeField="close"
             displayName="AAPL"
         >
         </mx:HLOCSeries>
      </mx:series>
   </mx:HLOCChart>
   <mx:Legend dataProvider="{myChart}"/>
</mx:Application>
```

You can define a stroke object using an MXML tag, and then bind that stroke object to the chart's renderer properties. For an example, see "Applying styles by binding tag definitions" on page 1715.

## Using strokes in ActionScript

You can instantiate and manipulate a Stroke object in ActionScript by using the mx.graphics.Stroke class. You can then use the `setStyle()` method to apply the Stroke object to the chart, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/ActionScriptStroke.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     import mx.graphics.Stroke;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month: "Jan", Profit: 2000, Expenses: 1500},
        {Month: "Feb", Profit: 1000, Expenses: 200},
        {Month: "Mar", Profit: 1500, Expenses: 500}
     ]);

     public function changeStroke(e:Event):void {
        var s:Stroke = new Stroke(0x001100,2);
        s.alpha = .5;
        s.color = 0x0000FF;
        har1.setStyle("axisStroke",s);
        var1.setStyle("axisStroke",s);
     }
  ]]></mx:Script>

  <mx:Stroke id="baseAxisStroke"
    color="0x884422"
    weight="10"
    alpha=".25"
    caps="square"
  />

  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>

        <mx:horizontalAxisRenderers>
           <mx:AxisRenderer id="har1">
                <mx:axisStroke>{baseAxisStroke}</mx:axisStroke>
           </mx:AxisRenderer>
        </mx:horizontalAxisRenderers>
        <mx:verticalAxisRenderers>
           <mx:AxisRenderer id="var1">
                <mx:axisStroke>{baseAxisStroke}</mx:axisStroke>
           </mx:AxisRenderer>
        </mx:verticalAxisRenderers>

        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
  <mx:Button id="b1"
    click="changeStroke(event)"
    label="Change Stroke"
  />
</mx:Application>
```

## Defining strokes for LineSeries and AreaSeries

Some chart series have more than one stroke-related style property. For LineSeries, you use the `stroke` style property to define a style for the chart item's renderer. You use the `lineStroke` property to define the stroke of the actual line segments.

The following example creates a thick blue line for the LineChart control's line segments, with large red boxes at each data point, which use the CrossItemRenderer object as their renderer:

```
<?xml version="1.0"?>
<!-- charts/LineSeriesStrokes.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Line Chart">
     <mx:LineChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries
            yField="Profit"
            displayName="Profit"
           >
            <mx:itemRenderer>
                <mx:Component>
                    <mx:CrossItemRenderer
                        scaleX="1.5"
                        scaleY="1.5"
                    />
                </mx:Component>
            </mx:itemRenderer>
            <mx:fill>
                <mx:SolidColor
                    color="0x0000FF"
                />
            </mx:fill>
            <mx:stroke>
                <mx:Stroke
                    color="0xFF0066"
                    alpha="1"
                />
            </mx:stroke>
            <mx:lineStroke>
                <mx:Stroke
                    color="0x33FFFF"
                    weight="5"
                    alpha=".8"
                />
            </mx:lineStroke>
          </mx:LineSeries>
          <mx:LineSeries
                yField="Expenses"
                displayName="Expenses"
          />
        </mx:series>
     </mx:LineChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

Similarly, with the AreaSeries class, you use the `stroke` property to set a style for the chart item's renderer. You use the `areaStroke` style property to define the stroke of the line that defines the area.

## Using fills with chart controls

When charting multiple data series, or just to improve the appearance of your charts, you can control the fill for each series in the chart or each item in a series. The fill lets you specify a pattern that defines how Flex draws the chart element. You can also use fills to specify the background colors of the chart or bands of background colors defined by the grid lines. Fills can be solid or can use linear and radial gradients. A *gradient* specifies a gradual color transition in the fill color.

You can use fills in the following ways:

- `fill` — Set the value of the `fill` property on a series to a single fill. This applies that fill to all chart items in that series. For example, all columns in a single series will have the same fill.
- `fills` — Set the value of the `fills` property on a series to an Array of fills. This applies a different fill from the Array to each chart item in the series. For example, each column in a series will have a different fill. The LineSeries and LineRenderer objects are not affected by the `fill` property's settings.

- `fillFunction` — Set the value of the `fillFunction` property on a series to point to a custom method. This method returns a fill based on the values of the chart item in the series. The return value of this function takes precedence over fills specified as styles. For information on using the `fillFunction` property, see "Using per-item fills" on page 1856.

All series except the HLOCSeries class support setting the fill-related properties.

If you use the `fills` property or the `fillFunction` to define the fills of chart items, and you want a legend, you must manually create the Legend object for that chart. For more information on creating Legend objects, see "Using Legend controls" on page 1873.

One of the most common uses of a fill is to control the color of the chart when you have multiple data series in a chart. The following example uses the `fill` property to set the color for each ColumnSeries object in a ColumnChart control:

```
<?xml version="1.0"?>
<!-- charts/ColumnFills.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           >
               <mx:fill>
                    <mx:SolidColor color="0x336699"/>
               </mx:fill>
           </mx:ColumnSeries>

           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           >
            <mx:fill>
                <mx:SolidColor color="0xFF99FF"/>
            </mx:fill>
           </mx:ColumnSeries>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

If you do not explicitly define different fills for multiple data series, Flex chooses solid colors for you.

You can also specify an array of colors using the `fills` property of the series. This property takes an Array of objects that define the fills. You can use this property to define a different color for each item in the series. If you do not define as many colors in the Array as there are items in the series, the chart starts with the first item in the Array on the next item.

The following example defines two Arrays of SolidColor objects. The first Array defines the colors of the items in the first series in the chart and the second Array defines the colors of the items in the second series. All of the fills in this example are partially transparent (the alpha value in the SolidColor constructor is .5).

```
<?xml version="1.0"?>
<!-- charts/SimpleFillsExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
    <![CDATA[
     import mx.graphics.SolidColor;
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);

     // Make all fills partially transparent by
     // setting the alpha to .5.
     [Bindable]
     private var warmColorsArray:Array = new Array(
        new SolidColor(0xFF0033, .5),
        new SolidColor(0xFF0066, .5),
        new SolidColor(0xFF0099, .5)
     );
     [Bindable]
     private var coolColorsArray:Array = new Array(
        new SolidColor(0x3333CC, .5),
        new SolidColor(0x3366CC, .5),
        new SolidColor(0x3399CC, .5)
     );

    ]]>
  </mx:Script>
  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
                fills="{warmColorsArray}"
```

```
          >
          </mx:ColumnSeries>

          <mx:ColumnSeries
               xField="Month"
               yField="Expenses"
               displayName="Expenses"
               fills="{coolColorsArray}"
          >
          </mx:ColumnSeries>
       </mx:series>
    </mx:ColumnChart>
  </mx:Panel>
</mx:Application>
```

This example does not have a legend. If you use the `fills` property or the `fillFunction` to define the fills of chart items, and you want a legend, you must manually create the Legend object for that chart. For more information on creating Legend objects, see "Using Legend controls" on page 1873.

With the PieSeries, you typically use a single Array of fills to specify how Flex should draw the individual wedges. For example, you can give each wedge that represents a PieSeries its own color, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/PieFills.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
    <![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Expense:"Taxes", Amount:2000},
        {Expense:"Rent", Amount:1000},
        {Expense:"Bills", Amount:100},
        {Expense:"Car", Amount:450},
        {Expense:"Gas", Amount:100},
        {Expense:"Food", Amount:200}
     ]);
    ]]>
  </mx:Script>
  <mx:Panel title="Pie Chart">
     <mx:PieChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:series>
           <mx:PieSeries
                field="Amount"
                nameField="Expense"
                labelPosition="callout"
           >
                <mx:fills>
                    <mx:SolidColor color="0xCC66FF" alpha=".8"/>
                    <mx:SolidColor color="0x9966CC" alpha=".8"/>
                    <mx:SolidColor color="0x9999CC" alpha=".8"/>
                    <mx:SolidColor color="0x6699CC" alpha=".8"/>
                    <mx:SolidColor color="0x669999" alpha=".8"/>
                    <mx:SolidColor color="0x99CC99" alpha=".8"/>
                </mx:fills>
            </mx:PieSeries>
        </mx:series>
     </mx:PieChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

This example also illustrates that you can use the `<mx:fills>` tag inside a series to define an Array of fills for that series.

You can also use fills to set the background of the charts. You do this by adding an `<mx:fill>` child tag to the chart tag, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/BackgroundFills.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Expense:"Taxes", Amount:2000},
        {Expense:"Rent", Amount:1000},
        {Expense:"Bills", Amount:100}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Background Fill">
     <mx:BarChart
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Expense"
           />
        </mx:verticalAxis>
        <mx:fill>
           <mx:SolidColor
                color="0x66CCFF"
                alpha=".5"
           />
        </mx:fill>
        <mx:series>
           <mx:BarSeries xField="Amount"/>
        </mx:series>
     </mx:BarChart>
  </mx:Panel>
</mx:Application>
```

## Setting fills with CSS

You can use the `fill` or `fills` style properties in an `<mx:Style>` declaration using CSS syntax. You can use either a type or class selector. The following example sets the fill of the custom `myBarChartStyle` class selector to `#FF0000`:

```
<?xml version="1.0"?>
<!-- charts/BackgroundFillsCSS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Expense:"Taxes", Amount:2000},
        {Expense:"Rent", Amount:1100},
        {Expense:"Bills", Amount:100}
     ]);
  ]]></mx:Script>

  <mx:Style>
     .myBarChartStyle {
        fill:#FF0000;
     }
  </mx:Style>

  <mx:Panel title="Background Fill">
     <mx:BarChart
        dataProvider="{expenses}"
        showDataTips="true"
        styleName="myBarChartStyle"
     >
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Expense"
           />
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries xField="Amount"/>
        </mx:series>
     </mx:BarChart>
  </mx:Panel>
</mx:Application>
```

Flex converts the fill style in the class selector to a SolidColor object.

The following example defines colors for each of the series by setting the value of the `fill` style property in the custom class selectors:

```
<?xml version="1.0"?>
<!-- charts/SimpleCSSFillsExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
    <![CDATA[
     import mx.graphics.SolidColor;
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);

    ]]>
  </mx:Script>

  <mx:Style>
        .myRedColumnSeries {
            fill:#FF0033;
        }
        .myGreenColumnSeries {
            fill:#33FF00;
        }
  </mx:Style>

  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
                styleName="myRedColumnSeries"
           >
           </mx:ColumnSeries>

           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
                styleName="myGreenColumnSeries"
           >
           </mx:ColumnSeries>
        </mx:series>
     </mx:ColumnChart>
  </mx:Panel>
</mx:Application>
```

To specify an Array for the `fills` property in CSS, you use a comma-separated list, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/CSSFillsArrayExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
    <![CDATA[
     import mx.graphics.SolidColor;
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);

    ]]>
  </mx:Script>

  <mx:Style>
        .myRedColumnSeries {
            fills: #FF0033, #FF3333, #FF6633;
        }
        .myGreenColumnSeries {
            fills: #33FF00, #33FF33, #33FF66;
        }
  </mx:Style>

  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
                styleName="myRedColumnSeries"
           >
           </mx:ColumnSeries>

           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
                styleName="myGreenColumnSeries"
            >
            </mx:ColumnSeries>
         </mx:series>
      </mx:ColumnChart>
  </mx:Panel>
</mx:Application>
```

## Using gradient fills with chart controls

Flex provides several classes that let you specify gradient fills. You use either the LinearGradient class or the RadialGradient class, along with the GradientEntry class to specify a gradient fill. The following table describes these classes:

| Class | Description |
|---|---|
| LinearGradient | Defines a gradient fill that starts at a boundary of the chart element. You specify an array of GradientEntry objects to control gradient transitions with the LinearGradient object. |
| RadialGradient | Defines a gradient fill that radiates from the center of a chart element. You specify an array of GradientEntry objects to control gradient transitions with the RadialGradient object. |
| GradientEntry | Defines the objects that control the gradient transition. Each GradientEntry object contains the following properties:<br>• `color` Specifies a color value.<br>• `alpha` Specifies the transparency. Valid values are 0 (invisible) through 1 (opaque). The default value is 1.<br>• `ratio` Specifies where in the chart, as a percentage, Flex starts the transition to the next `color`. For example, if you set the `ratio` property to .33, Flex begins the transition 33% of the way through the chart. If you do not set the `ratio` property, Flex tries to evenly apply values based on the `ratio` properties for the other GradientEntry objects. Valid values range from 0 to 1. |

The following example uses a LinearGradient class with three colors for a gradient fill of the chart's background:

```
<?xml version="1.0"?>
<!-- charts/GradientFills.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Expense:"Taxes", Amount:2000},
        {Expense:"Rent", Amount:1000},
        {Expense:"Bills", Amount:100}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Background Fill">
     <mx:BarChart
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:verticalAxis>
```

```
            <mx:CategoryAxis
                 dataProvider="{expenses}"
                 categoryField="Expense"
            />
         </mx:verticalAxis>
         <mx:fill>
            <mx:LinearGradient>
             <mx:entries>
                 <mx:GradientEntry
                     color="0xC5C551"
                     ratio="0"
                     alpha="1"
                 />
                 <mx:GradientEntry
                     color="0xFEFE24"
                     ratio=".33"
                     alpha="1"
                 />
                 <mx:GradientEntry
                     color="0xECEC21"
                     ratio=".66"
                     alpha="1"
                 />
             </mx:entries>
            </mx:LinearGradient>
         </mx:fill>
         <mx:series>
            <mx:BarSeries xField="Amount"/>
         </mx:series>
      </mx:BarChart>
  </mx:Panel>
</mx:Application>
```

The LinearGradient object takes a single attribute, `angle`. By default, it defines a transition from left to right across the chart. Use the `angle` property to control the direction of the transition. For example, a value of 180 causes the transition to occur from right to left, rather than from left to right.

The following example sets the `angle` property to 90, which specifies that the transition occurs from the top of the chart to the bottom.

```
<?xml version="1.0"?>
<!-- charts/GradientFillsAngled.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Expense:"Taxes", Amount:2000},
        {Expense:"Rent", Amount:1000},
        {Expense:"Bills", Amount:100}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Background Fill">
     <mx:BarChart dataProvider="{expenses}" showDataTips="true">
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Expense"
           />
        </mx:verticalAxis>
        <mx:fill>
           <mx:LinearGradient angle="90">
            <mx:entries>
                <mx:GradientEntry
                    color="0xC5C551"
                    ratio="0"
                    alpha="1"
                />
                <mx:GradientEntry
                    color="0xFEFE24"
                    ratio=".33"
                    alpha="1"
                />
                <mx:GradientEntry
                    color="0xECEC21"
                    ratio=".66"
                    alpha="1"
                />
            </mx:entries>
           </mx:LinearGradient>
        </mx:fill>
        <mx:series>
           <mx:BarSeries xField="Amount"/>
        </mx:series>
     </mx:BarChart>
  </mx:Panel>
```

```
</mx:Application>
```

## Using different alpha values with fills

When charting multiple data series, you can define the series to overlap. For example, the column chart lets you display the columns next to each other or overlap them for multiple data series. To overlap series, you set the `type` property of the chart to `overlaid`. The same is true for an area series.

When you have multiple data series that overlap, you can specify that the fill for each series has an alpha value less than 100%, so that the series have a level of transparency. The valid values for the `alpha` property are 0 (invisible) through 1 (opaque).

You cannot specify alpha values for fills if you apply the fills using CSS.

The following example defines an area chart in which each series in the chart uses a solid fill with the same level of transparency:

```
<?xml version="1.0"?>
<!-- charts/AlphaFills.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Area Chart">
     <mx:AreaChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:AreaSeries
                yField="Profit"
                displayName="Profit"
           >
                <mx:areaStroke>
                    <mx:Stroke
                        color="0x9A9A00"
                        weight="2"
                    />
                </mx:areaStroke>
                <mx:areaFill>
                    <mx:SolidColor
                        color="0x7EAEFF"
                        alpha=".3"
                    />
                </mx:areaFill>
           </mx:AreaSeries>
           <mx:AreaSeries
                yField="Expenses"
                displayName="Expenses"
           >
                <mx:areaStroke>
```

```
                    <mx:Stroke
                        color="0x9A9A00"
                        weight="2"
                    />
                </mx:areaStroke>
                <mx:areaFill>
                    <mx:SolidColor
                        color="0xAA0000"
                        alpha=".3"
                    />
                </mx:areaFill>
           </mx:AreaSeries>
        </mx:series>
     </mx:AreaChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

If you define a gradient fill, you can set the `alpha` property on each entry in the Array of `<mx:GradientEntry>` tags, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/GradientAlphaFills.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Area Chart">
     <mx:AreaChart id="myChart" dataProvider="{expenses}"
     showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:AreaSeries yField="Profit" displayName="Profit">
            <mx:areaStroke>
                <mx:Stroke color="0x9A9A00" weight="2"/>
            </mx:areaStroke>
            <mx:areaFill>
                <mx:LinearGradient angle="90">
                    <mx:entries>
                        <mx:GradientEntry
                            color="0xC5C551"
                            ratio="0"
                            alpha="1"
                        />
                        <mx:GradientEntry
                            color="0xFEFE24"
                            ratio=".33"
                            alpha="1"
                        />
                        <mx:GradientEntry
                            color="0xECEC21"
                            ratio=".66"
                            alpha=".2"
                        />
                    </mx:entries>
                </mx:LinearGradient>
            </mx:areaFill>
```

```
          </mx:AreaSeries>
          <mx:AreaSeries
           yField="Expenses"
           displayName="Expenses"
          >
           <mx:areaStroke>
               <mx:Stroke color="0x9A9A00" weight="2"/>
           </mx:areaStroke>
           <mx:areaFill>
               <mx:LinearGradient angle="90">
                   <mx:entries>
                       <mx:GradientEntry
                           color="0xAA0000"
                           ratio="0"
                           alpha="1"
                       />
                       <mx:GradientEntry
                           color="0xCC0000"
                           ratio=".33"
                           alpha="1"
                       />
                       <mx:GradientEntry
                           color="0xFF0000"
                           ratio=".66"
                           alpha=".2"
                       />
                   </mx:entries>
               </mx:LinearGradient>
           </mx:areaFill>
          </mx:AreaSeries>
       </mx:series>
    </mx:AreaChart>
    <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

In this example, you make the last gradient color in the series fully transparent by setting its `alpha` property to 0.

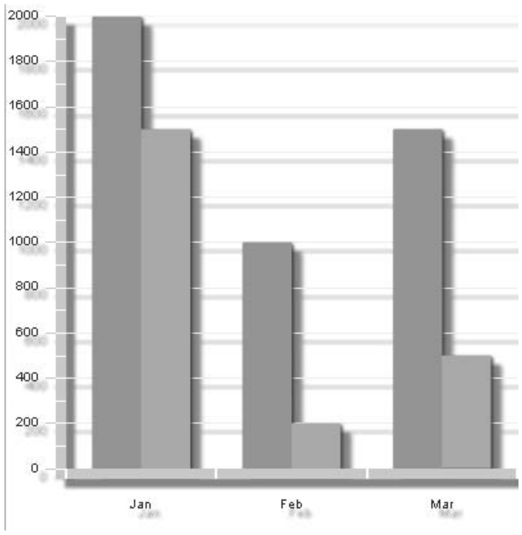
## Using filters with chart controls

You can add filters such as drop shadows to your charts by using the classes in the flash.filters package. These filters include:

- BevelFilter
- BitmapFilter
- BlurFilter

- ColorMatrixFilter
- DisplacementMapFilter
- DropShadowFilter
- GlowFilter
- GradientBevelFilter
- GradientGlowFilter

You can apply filters to the chart control itself, or to each chart series. When you apply a filter to a chart control, the filter is applied to all aspects of that chart control, including gridlines, axis labels, and each data point in the series. The following image shows a drop shadow filter applied to a ColumnChart control:



The following example applies a custom drop shadow filter to a ColumnChart control. The result is that every element, including the grid lines, axis labels, and columns, has a background shadow.

```
<?xml version="1.0"?>
<!-- charts/ColumnWithDropShadow.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:flash="flash.filters.*">

  <mx:Script><![CDATA[
  import mx.collections.ArrayCollection;
  [Bindable]
  public var expenses:ArrayCollection = new ArrayCollection([
     {Month:"Jan", Profit:2000, Expenses:1500},
     {Month:"Feb", Profit:1000, Expenses:200},
     {Month:"Mar", Profit:1500, Expenses:500}
  ]);
  ]]></mx:Script>
  <mx:Panel>
     <mx:ColumnChart id="column" dataProvider="{expenses}">
        <!-- Add a custom drop shadow filter to the
              ColumnChart control. -->
        <mx:filters>
           <flash:DropShadowFilter
                distance="10"
                color="0x666666"
                alpha=".8"
           />
        </mx:filters>
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{column}"/>
  </mx:Panel>
```

```
</mx:Application>
```

When you use a filter in your Flex application, ensure that you add the flash.filters namespace to your MXML file's top-level tag, as the following example shows:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:flash="flash.filters.*">
```

For more information on using filters, see "Using filters in Flex" on page 824.

Adding a drop shadow filter to some chart controls can have unexpected consequences. For example, if you add a drop shadow filter to a PieChart control, Flex renders that drop shadow filter in addition to the drop shadow filter that it applies to the PieSeries by default.

You can remove filters by setting the filters Array to an empty Array, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/ClearFilters.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
  import mx.collections.ArrayCollection;
  [Bindable]
  public var expenses:ArrayCollection = new ArrayCollection([
     {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
     {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
     {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
  ]);
  ]]></mx:Script>
  <mx:LineChart id="myChart" dataProvider="{expenses}">
     <mx:seriesFilters>
        <mx:Array/>
     </mx:seriesFilters>
     <mx:horizontalAxis>
        <mx:CategoryAxis
            dataProvider="{expenses}"
            categoryField="Month"
        />
     </mx:horizontalAxis>
     <mx:series>
        <mx:LineSeries
            yField="Profit"
            displayName="Profit"
        />
        <mx:LineSeries
            yField="Expenses"
            displayName="Expenses"
        />
        <mx:LineSeries
            yField="Amount"
            displayName="Amount"
        />
     </mx:series>
  </mx:LineChart>
</mx:Application>
```

The following example creates a PieChart control and applies a drop shadow to it; it also removes the default drop shadow filter from the PieSeries so that there is a single drop shadow:

```
<?xml version="1.0"?>
<!-- charts/PieChartShadow.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns:flash=
"flash.filters.*">
  <mx:Script>
    <![CDATA[
     [Bindable]
     public var expenses:Object = [
        {Expense:"Taxes", Amount:2000},
        {Expense:"Gas", Amount:100},
        {Expense:"Food", Amount:200}
     ];
    ]]>
  </mx:Script>
  <mx:Panel title="Pie Chart">
     <mx:PieChart id="pie"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <!-- Add a custom drop shadow to the PieChart control -->
        <mx:filters>
           <flash:DropShadowFilter
                distance="10"
                color="0x666666"
                alpha=".8"
           />
        </mx:filters>
        <mx:series>
           <mx:Array>
            <mx:PieSeries field="Amount" nameField="Expense"
            labelPosition="callout" explodeRadius=".2">
                <!-- Clear default shadow on the PieSeries -->
                <mx:filters>
                    <mx:Array/>
                </mx:filters>
            </mx:PieSeries>
           </mx:Array>
        </mx:series>
     </mx:PieChart>
     <mx:Legend dataProvider="{pie}"/>
  </mx:Panel>
</mx:Application>
```

For more information on working with grid lines, see “Using grid lines” on page 1768.

You can add filters to individual chart elements that are display objects, such as series, grid lines, legend items, and axes. The following example defines set of filters, and then applies them to various chart elements:

```
<?xml version="1.0"?>
<!-- charts/MultipleFilters.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns:flash=
"flash.filters.*" creationComplete="createFilters()">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);

     private var myBlurFilter:BlurFilter;
     private var myGlowFilter:GlowFilter;
     private var myBevelFilter:BevelFilter;
     private var myDropShadowFilter:DropShadowFilter;

     private var color:Number = 0xFF33FF;

     public function applyFilters():void {
        // Apply filters to series, grid lines, legend, and axis.
        myGridlines.filters = [myBlurFilter];
        myLegend.filters = [myGlowFilter];
        myAxisRenderer.filters = [myBevelFilter];
        s1.filters = [myDropShadowFilter];
        s2.filters = [myDropShadowFilter];
     }

     public function createFilters():void {
        // Define filters.
        myBlurFilter = new BlurFilter(4,4,1);

        myGlowFilter = new GlowFilter(color, .8, 6, 6,
            2, 1, false, false);

        myDropShadowFilter = new DropShadowFilter(15, 45,
            color, 0.8, 8, 8, 0.65, 1, false, false);

        myBevelFilter = new BevelFilter(5, 45, color, 0.8,
           0x333333, 0.8, 5, 5, 1, BitmapFilterQuality.HIGH,
            BitmapFilterType.INNER, false);

        applyFilters();
     }
  ]]></mx:Script>
```

```
<mx:Panel>
   <mx:ColumnChart id="myChart" dataProvider="{expenses}">
      <mx:backgroundElements>
         <mx:GridLines id="myGridlines"
              horizontalChangeCount="1"
             verticalChangeCount="1"
             direction="both"
         >
         </mx:GridLines>
      </mx:backgroundElements>
      <mx:horizontalAxis>
         <mx:CategoryAxis
              dataProvider="{expenses}"
              categoryField="Month"
         />
      </mx:horizontalAxis>

      <mx:horizontalAxisRenderers>
         <mx:AxisRenderer id="myAxisRenderer"
              placement="bottom"
             canDropLabels="true"
         >
          <mx:axisStroke>
              <mx:Stroke color="#000080" weight="10"/>
          </mx:axisStroke>
          <mx:tickStroke>
              <mx:Stroke color="#000060" weight="5"/>
          </mx:tickStroke>
          <mx:minorTickStroke>
              <mx:Stroke color="#100040" weight="5"/>
          </mx:minorTickStroke>
         </mx:AxisRenderer>
      </mx:horizontalAxisRenderers>

      <mx:series>
         <mx:ColumnSeries id="s1"
              xField="Month"
              yField="Profit"
              displayName="Profit"
         />
         <mx:ColumnSeries id="s2"
              xField="Month"
              yField="Expenses"
              displayName="Expenses"
         />
      </mx:series>
   </mx:ColumnChart>
   <mx:Legend  id="myLegend" dataProvider="{myChart}"/>
</mx:Panel>
```

```
</mx:Application>
```

## Using grid lines

All charts except the PieChart control have grid lines by default. You can control those grid lines with the CSS `gridLinesStyleName` property, and with the chart series' `backgroundElements` and `annotationElements` properties.

You can include horizontal, vertical, or both grid lines in your chart with the GridLines object. You can set these behind the data series by using the chart's `backgroundElements` property or in front of the data series by using the `annotationElements` property.

The following example turns on grid lines in both directions and applies them to the chart:

```
<?xml version="1.0"?>
<!-- charts/GridLinesBoth.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);
  ]]></mx:Script>

  <mx:Array id="bge">
     <mx:GridLines direction="both"/>
  </mx:Array>

  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart"
        dataProvider="{expenses}"
        backgroundElements="{bge}"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
```

```
            />
         </mx:series>
      </mx:ColumnChart>
      <mx:Legend dataProvider="{myChart}"/>
   </mx:Panel>
</mx:Application>
```

NOTE: The `annotationElements` property refers to any chart elements that appear in the foreground of your chart, and the `backgroundElements` property refers to any chart elements that appear behind the chart's data series.

You can also define the grid lines inside each chart control's definition, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/GridLinesBothInternal.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);
  ]]></mx:Script>

  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:backgroundElements>
           <mx:GridLines direction="both"/>
        </mx:backgroundElements>

        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
```

```
    </mx:ColumnChart>
    <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

To define the fills and strokes for grid lines, you use the `horizontalStroke`, `verticalStroke`, `horizontalFill`, and `verticalFill` properties. The following properties also define the appearance of grid lines:

- `horizontalAlternateFill`
- `horizontalChangeCount`
- `horizontalOriginCount`
- `horizontalShowOrigin`
- `horiztontalTickAligned`
- `verticalAlternateFill`
- `verticalChangeCount`
- `verticalOriginCount`
- `verticalShowOrigin`
- `verticalTickAligned`

For information on working with strokes, see "Using strokes" on page 1731. For more information on using the `backgroundElements` and `annotationElements` properties, see "Using ChartElement objects" on page 1718.

You can manipulate the appearance of the grid lines directly in MXML, with ActionScript, or with CSS. The following sections describe techniques for formatting grid lines for Flex Charting objects.

## Formatting grid lines with MXML

To control the appearance of the grid lines, you can specify an array of GridLines objects as MXML tags, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/GridLinesFormatMXML.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);
  ]]></mx:Script>

  <mx:Array id="bge">
     <mx:GridLines
        horizontalChangeCount="1"
        verticalChangeCount="1"
        direction="both"
     >
        <mx:horizontalStroke>
           <mx:Stroke weight="3"/>
        </mx:horizontalStroke>
        <mx:verticalStroke>
           <mx:Stroke weight="3"/>
        </mx:verticalStroke>
        <mx:horizontalFill>
           <mx:SolidColor color="0x99033" alpha=".66"/>
        </mx:horizontalFill>
     </mx:GridLines>
  </mx:Array>

  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart"
        dataProvider="{expenses}"
        backgroundElements="{bge}"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

This example uses the `changeCount` property to specify that Flex draws grid lines at every tick mark along the axis, and sets the `direction` property to `both`. This causes Flex to draw grid lines both horizontally and vertically. You could also specify `horizontal` or `vertical` as values for the `direction` property.

To remove grid lines entirely, you can set the `backgroundElements` property to an empty Array, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/NoGridLines.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;
            [Bindable]
            private var s1:ArrayCollection = new ArrayCollection( [
                {"x": 20, "y": 10, "r1":10 },
                {"x": 40, "y": 5, "r1":20 } ,
                {"x": 60, "y": 0, "r1":30 }]);

    ]]>
    </mx:Script>

    <mx:Array id="bge">
    </mx:Array>

    <mx:BubbleChart id="bc"
        showDataTips="true"
        backgroundElements="{bge}"
    >
        <mx:series>
            <mx:BubbleSeries
                dataProvider="{s1}"
                displayName="series1"
                xField="x"
                yField="y"
                radiusField="r1"
            />
        </mx:series>
    </mx:BubbleChart>
</mx:Application>
```

You can also change the appearance of grid lines by using filters such as a drop shadow, glow, or bevel. For more information, see “Using filters with chart controls” on page 1759.

## Formatting grid lines with CSS

You can set the style of grid lines by applying a CSS style to the GridLines object. The following example applies the `myStyle` style to the grid lines:

```
<?xml version="1.0"?>
<!-- charts/GridLinesFormatCSS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);
  ]]></mx:Script>

  <mx:Style>
     .myStyle {
        direction:"both";
        horizontalShowOrigin:true;
        horizontalTickAligned:false;
        horizontalChangeCount:1;
        verticalShowOrigin:false;
        verticalTickAligned:true;
        verticalChangeCount:1;
        horizontalFill:#990033;
        horizontalAlternateFill:#00CCFF;
     }
  </mx:Style>

  <mx:Array id="bge">
     <mx:GridLines styleName="myStyle">
        <mx:horizontalStroke>
           <mx:Stroke weight="3"/>
        </mx:horizontalStroke>
        <mx:verticalStroke>
           <mx:Stroke weight="3"/>
        </mx:verticalStroke>
     </mx:GridLines>
  </mx:Array>

  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart"
        dataProvider="{expenses}"
        backgroundElements="{bge}"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
```

```
                dataProvider="{expenses}"
                categoryField="Month"
             />
          </mx:horizontalAxis>
          <mx:series>
             <mx:ColumnSeries
                  xField="Month"
                  yField="Profit"
                  displayName="Profit"
             />
             <mx:ColumnSeries
                  xField="Month"
                  yField="Expenses"
                  displayName="Expenses"
             />
          </mx:series>
       </mx:ColumnChart>
       <mx:Legend dataProvider="{myChart}"/>
    </mx:Panel>
</mx:Application>
```

## Formatting grid lines with ActionScript

You can manipulate the GridLines at run time with ActionScript. The following example adds filled grid lines in front of and behind the chart's series:

```
<?xml version="1.0"?>
<!-- charts/GridLinesFormatActionScript.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.graphics.SolidColor;
     import mx.graphics.Stroke;
     import mx.charts.GridLines;
     import mx.collections.ArrayCollection;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);

     [Bindable]
     public var bge:GridLines;

     public function addGridLines():void {
        bge = new GridLines();

        var s:Stroke = new Stroke(0xff00ff, 2);
        bge.setStyle("horizontalStroke", s);

        var f:SolidColor = new SolidColor(0x990033, .3);
        bge.setStyle("horizontalFill",f);

        var f2:SolidColor = new SolidColor(0x336699, .3);
        bge.setStyle("horizontalAlternateFill",f2);

        myChart.backgroundElements = [bge];
     }
  ]]></mx:Script>

  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart"
        dataProvider="{expenses}"
        creationComplete="addGridLines()"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

## Positioning the axes

You can place axes on the left, right, top, or bottom of the chart control. This includes the axis line as well as labels and any tick marks you have added. You can place horizontal axes at the top or bottom of the chart (or both, if you have multiple vertical axes). You can place vertical axes at the left or right of the chart (or both, if you have multiple horizontal axes).

To change the location of axes, you use the `placement` property of the AxisRenderer. Valid values for this property for a vertical axis are `top` and `bottom`. Valid values for this property for a horizontal axis are `left` and `right`.

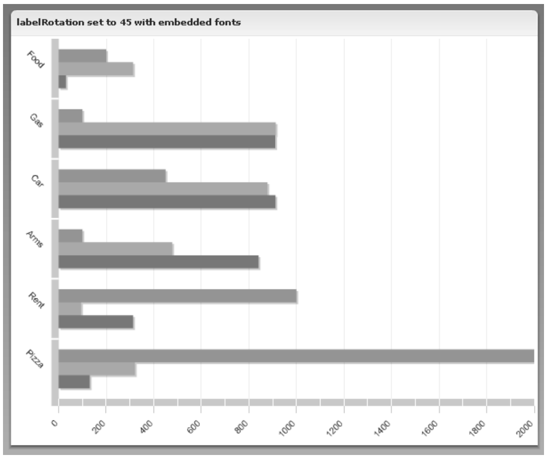
The following example creates a ColumnChart control with the default axis locations (bottom and left). You can select new locations by using the ComboBox controls at the bottom of the panel.

```
<?xml version="1.0"?>
<!-- charts/AxisPlacementExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);

     [Bindable]
     private var horChoices:Array = ["bottom","top"];

     [Bindable]
     private var vertChoices:Array = ["left","right"];

  ]]></mx:Script>
  <mx:Panel title="Variable Axis Placement">
     <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:horizontalAxisRenderers>
            <mx:AxisRenderer id="horAxisRend"
                axis="{axis1}"
                placement="{myHorBox.selectedItem}"
            />
        </mx:horizontalAxisRenderers>

        <mx:verticalAxisRenderers>
            <mx:AxisRenderer id="vertAxisRend"
                axis="{axis2}"
                placement="{myVertBox.selectedItem}"
            />
        </mx:verticalAxisRenderers>

        <mx:horizontalAxis>
           <mx:CategoryAxis id="axis1"
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>

        <mx:verticalAxis>
            <mx:LinearAxis id="axis2"/>
        </mx:verticalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>

    <mx:Form>
        <mx:FormItem label="Horizontal Axis Location:">
            <mx:ComboBox id="myHorBox" dataProvider="{horChoices}"/>
        </mx:FormItem>
        <mx:FormItem label="Vertical Axis Location:">
            <mx:ComboBox id="myVertBox" dataProvider="{vertChoices}"/>
        </mx:FormItem>
    </mx:Form>

  </mx:Panel>
</mx:Application>
```

## Rotating axis labels

You can rotate axis labels using the `labelRotation` property of the AxisRenderer object. You specify a number from -90 to 90, in degrees. If you set the `labelRotation` property to `null`, Flex determines an optimal angle and renders the axis labels.

The following example shows both sets of axis labels rotated 45 degrees:



To rotate axis labels, you must embed the font in the Flex application. If you rotate the the axis labels without embedding a font, they are rendered horizontally.

The following `<mx:Style>` block embeds a font in the Flex application, and then applies that font to the chart control that rotates its horizontal and vertical axis labels 45 degrees:

```
<?xml version="1.0"?>
<!-- charts/RotateAxisLabels.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
  import mx.collections.ArrayCollection;
  [Bindable]
  public var expenses:ArrayCollection = new ArrayCollection([
     {Month: "Jan", Profit: 2000, Expenses: 1500},
     {Month: "Feb", Profit: 1000, Expenses: 200},
     {Month: "Mar", Profit: 1500, Expenses: 500}
  ]);
  ]]></mx:Script>

  <mx:Style>
     @font-face{
        src: url("../assets/MyriadWebPro.ttf");
        fontFamily: myMyriad;
     }

     ColumnChart {
        fontFamily: myMyriad;
        fontSize: 20;
     }
  </mx:Style>

  <mx:Panel>
     <mx:ColumnChart id="column" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
                title="FY 2006"
           />
        </mx:horizontalAxis>

        <mx:horizontalAxisRenderers>
           <mx:AxisRenderer labelRotation="45"/>
        </mx:horizontalAxisRenderers>
        <mx:verticalAxisRenderers>
           <mx:AxisRenderer labelRotation="45"/>
        </mx:verticalAxisRenderers>

        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
               displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
               displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{column}"/>
  </mx:Panel>
</mx:Application>
```

## Skinning ChartItem objects

A ChartItem object represents a data point in a series. There is one ChartItem instance for each item in the series's data provider. ChartItem objects contain details about the data for the data point as well as the renderer (or, *skin*) to use when rendering that data point in the series. The ChartItem renderers define objects such as the icon that represents a data point in a PlotChart control or the box that makes up a bar in a BarChart control.

Each series has a default renderer that Flex uses to draw that series's ChartItem objects. You can specify a new renderer to use with the series's `itemRenderer` style property. This property points to a class that defines the appearance of the ChartItem object.

You can use existing classes to change the default renderers of chart items. The DiamondItemRenderer class is the default renderer for ChartItem objects in a data series in a PlotChart control. The following example uses the default DiamondItemRenderer class for the first data series. The second series uses the CircleItemRenderer class, which draws a circle to represent the data points in that series. The third series uses the CrossItemRenderer class, which draws a cross shape to represent the data points in that series.

```
<?xml version="1.0"?>
<!-- charts/PlotRenderers.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"January", Profit:2000, Expenses:1500, Amount:450},
        {Month:"February", Profit:1000, Expenses:200, Amount:600},
        {Month:"March", Profit:1500, Expenses:500, Amount:300},
        {Month:"April", Profit:500, Expenses:300, Amount:500},
        {Month:"May", Profit:1000, Expenses:450, Amount:250},
        {Month:"June", Profit:2000, Expenses:500, Amount:700}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Plot Chart">
     <mx:PlotChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:series>
           <!-- First series uses default renderer. -->
           <mx:PlotSeries
                xField="Expenses"
                yField="Profit"
                displayName="Plot 1"
           />

           <!-- Second series uses DiamondItemRenderer. -->
```

```
            <mx:PlotSeries
                 xField="Amount"
                 yField="Expenses"
                 displayName="Plot 2"
                 itemRenderer="mx.charts.renderers.CircleItemRenderer"
            />

            <!-- Third series uses CrossItemRenderer. -->
            <mx:PlotSeries
                 xField="Profit"
                 yField="Amount"
                 displayName="Plot 3"
                 itemRenderer="mx.charts.renderers.CrossItemRenderer"
            />
         </mx:series>
      </mx:PlotChart>
      <mx:Legend dataProvider="{myChart}"/>
   </mx:Panel>
</mx:Application>
```

## Using multiple renderer classes

You can sometimes choose from more than one renderer for a chart series, depending on the series. These renderers let you change the appearance of your charts by adding shadows or graphics to the chart items.

The following table lists the available renderer classes for the ChartItem objects of each chart type:

| Chart type | Available renderer classes |
|---|---|
| AreaChart | AreaRenderer |
| BarChart<br>BubbleChart<br>ColumnChart<br>PlotChart | BoxItemRenderer<br>CircleItemRenderer<br>CrossItemRenderer<br>DiamondItemRenderer<br>ShadowBoxItemRenderer<br>TriangleItemRenderer |
| CandlestickChart<br>HLOCChart | CandlestickItemRenderer<br>HLOCItemRenderer |
| LineChart | LineRenderer<br>ShadowLineRenderer |
| PieChart | WedgeItemRenderer |

The appearance of most renderers is self-explanatory. The BoxItemRenderer class draws ChartItem objects in the shape of boxes. The DiamondItemRenderer class draws ChartItem objects in the shape of diamonds. The ShadowBoxItemRenderer and ShadowLineRenderer classes add shadows to the ChartItem objects that they draw.

Some series types require multiple renderers to completely render their data. For example, a LineSeries object has both an `itemRenderer` style property and a `lineSegmentRenderer` style property. The `itemRenderer` property specifies the renderer for the data items. The `lineSegmentRenderer` specifies the appearance of the line segments between items.

The other series type that requires two renderers is the AreaSeries. The `areaRenderer` property specifies the appearance of the area, and the `itemRenderer` specifies the appearance of the data items.

You can also specify the renderer to use for legends. The default is the class that the series' `itemRenderer` property specifies. For more information, see "Formatting Legend controls" on page 1791.

You can use multiple types of data series in a single chart. For example, you can use a ColumnSeries and a LineSeries to show something like a moving average over a stock price. In this case, you can use all the renderers supported by those series in the same chart. For more information on using multiple series, see "Using multiple data series" on page 1696.

## Creating custom renderers

You can replace the `itemRenderer` property of a chart series with a custom renderer. You define the renderer on the `itemRenderer` style property for the chart series. This renderer can be a graphical renderer or a class that programmatically defines the renderer.

### Creating graphical renderers

You can use a graphic file such as a GIF or JPEG to be used as a renderer on the chart series. You do this by setting the value of the `itemRenderer` style property to be an embedded image. This method of graphically rendering chart items is similar to the graphical skimming method used for other components, as described in "Creating graphical skins" on page 899.

The following example uses the graphic file to represent data points on a PlotChart control:

```
<?xml version="1.0"?>
<!-- charts/CustomPlotRenderer.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"January", Profit:2000, Expenses:1500, Amount:450},
        {Month:"February", Profit:1000, Expenses:200, Amount:600},
        {Month:"March", Profit:1500, Expenses:500, Amount:300},
        {Month:"April", Profit:500, Expenses:300, Amount:500},
        {Month:"May", Profit:1000, Expenses:450, Amount:250},
        {Month:"June", Profit:2000, Expenses:500, Amount:700}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Plot Chart">
     <mx:PlotChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:series>

           <!-- First series uses embedded image for renderer. -->
           <mx:PlotSeries
                xField="Expenses"
                yField="Profit"
                displayName="Plot 1"
                itemRenderer="@Embed('../assets/bird.gif')"
                radius="50"
                legendMarkerRenderer="@Embed('../assets/bird.gif')"
           />

           <!-- Second series uses DiamondItemRenderer. -->
           <mx:PlotSeries
                xField="Amount"
                yField="Expenses"
                displayName="Plot 2"
                itemRenderer="mx.charts.renderers.CircleItemRenderer"
           />

           <!-- Third series uses CrossItemRenderer. -->
           <mx:PlotSeries
                xField="Profit"
                yField="Amount"
                displayName="Plot 3"
                itemRenderer="mx.charts.renderers.CrossItemRenderer"
           />
        </mx:series>
     </mx:PlotChart>
```

```
      <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

This example uses the bird.gif graphic to represent each data point on the plot chart. It controls the size of the embedded image by using the `radius` property.

You are not required to set the value of the `itemRenderer` property inline. You can also embed a graphic file in ActionScript as a Class, cast it to a ClassFactory, and then reference it inline, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/CustomPlotRendererAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"January", Profit:2000, Expenses:1500, Amount:450},
        {Month:"February", Profit:1000, Expenses:200, Amount:600},
        {Month:"March", Profit:1500, Expenses:500, Amount:300},
        {Month:"April", Profit:500, Expenses:300, Amount:500},
        {Month:"May", Profit:1000, Expenses:450, Amount:250},
        {Month:"June", Profit:2000, Expenses:500, Amount:700}
     ]);

     [Bindable]
     [Embed(source="../assets/bird.gif")]
     public var myBird:Class;

     [Bindable]
     public var myBirdFactory:ClassFactory =
        new ClassFactory(myBird);

  ]]></mx:Script>
  <mx:Panel title="Plot Chart">
     <mx:PlotChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:series>

           <!-- First series uses ActionScript class renderer. -->
           <mx:PlotSeries
                xField="Expenses"
                yField="Profit"
                displayName="Plot 1"
                itemRenderer="{myBirdFactory}"
                legendMarkerRenderer="{myBirdFactory}"
                radius="50"
           />
```

```
            <!-- Second series uses DiamondItemRenderer. -->
            <mx:PlotSeries
                 xField="Amount"
                 yField="Expenses"
                 displayName="Plot 2"
                 itemRenderer="mx.charts.renderers.CircleItemRenderer"
            />

            <!-- Third series uses CrossItemRenderer. -->
            <mx:PlotSeries
                 xField="Profit"
                 yField="Amount"
                 displayName="Plot 3"
                 itemRenderer="mx.charts.renderers.CrossItemRenderer"
            />
         </mx:series>
      </mx:PlotChart>
      <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

## Creating programmatic renderers

Creating a custom renderer class for your chart items can give you more control than creating simple graphical renderers. Using class-based renderers is very similar to using programmatic skins, as described in "Creating programmatic skins" on page 903.

One approach to is to extend the ProgrammaticSkin class and implement the IDataRenderer interface. In this approach, you can provide all of the logic for drawing chart items in your custom class, and maintain the greatest control over its appearance. For example, you use methods in the Graphics class to draw and fill the rectangles of the bars in a BarChart control.

When you implement the IDataRenderer interface, you must define a setter and getter method to implement the `data` property. This `data` property is of the type of the series item. In the case of a ColumnSeries, it is a ColumnSeriesItem. Other item types include BarSeriesItem, BubbleSeriesItem, LineSeriesItem, and PlotSeriesItem.

In your class, you override the `updateDisplayList()` method with the logic for drawing the chart item as well as setting any custom properties. You should also call the `super.updateDisplayList()` method.

The following example renders the chart items and uses an Array of colors to color each column in the ColumnChart control differently:

```
// charts/CycleColorRenderer.as

package { // Empty package.

  import mx.charts.series.items.ColumnSeriesItem;
  import mx.skins.ProgrammaticSkin;
  import mx.core.IDataRenderer;
  import flash.display.Graphics;

  public class CycleColorRenderer extends mx.skins.ProgrammaticSkin
     implements IDataRenderer {

     private var colors:Array = [0xCCCC99,0x999933,0x999966];
     private var _chartItem:ColumnSeriesItem;

     public function CycleColorRenderer() {
        // Empty constructor.
     }

     public function get data():Object {
        return _chartItem;
     }

     public function set data(value:Object):void {
        _chartItem = value as ColumnSeriesItem;
        invalidateDisplayList();
     }

     override protected function
        updateDisplayList(unscaledWidth:Number,unscaledHeight:Number):void
{
           super.updateDisplayList(unscaledWidth, unscaledHeight);
           var g:Graphics = graphics;
           g.clear();
           g.beginFill(colors[(_chartItem == null)? 0:_chartItem.index]);
           g.drawRect(0, 0, unscaledWidth, unscaledHeight);
           g.endFill();
     }
  } // Close class.
} // Close package.
```

In your Flex application, you use this class as the renderer by using the `itemRenderer` property of the ColumnSeries, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/ProgrammaticRenderer.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
    <![CDATA[
        [Bindable]
        public var expenses:Object = [
           {Month:"Jan", Profit:2000, Expenses:1500},
           {Month:"Feb", Profit:1000, Expenses:200},
           {Month:"Mar", Profit:1500, Expenses:500}
        ];
    ]]>
  </mx:Script>

  <mx:Panel>
     <mx:ColumnChart id="column" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:Array>
            <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
                itemRenderer="CycleColorRenderer"
            />
           </mx:Array>
        </mx:series>
     </mx:ColumnChart>
  </mx:Panel>
</mx:Application>
```

For more information on overriding the `updateDisplayList()` method, see “Implementing the updateDisplayList() method” on page 908.

## Formatting Legend controls

The Legend control is a subclass of the Tile class. You can use Tile properties and some properties of the Container class to format the Legend control. Also, the Legend control has properties (such as `labelPlacement`, `markerHeight`, and `markerWidth)` that you can use to format its appearance. For information on creating Legend controls, see "Using Legend controls" on page 1873.

The following table describes the Legend control properties:

| Property | Type | Description |
|---|---|---|
| `labelPlacement` | String | Specifies the alignment of the LegendItem object's label. Valid values are `right`, `left`, `top`, and `bottom`. |
| `markerHeight` | Number | Specifies the height, in pixels, of the LegendItem object's marker. |
| `markerWidth` | Number | Specifies the width, in pixels, of the LegendItem object's marker. |
| `renderer` | Object | Specifies a class for the LegendItem object's marker. The renderer must implement the IBoxRenderer interface. |
| `stroke` | Object | Specifies the line stroke for the LegendItem object's marker. For more information on defining line strokes, see "Using strokes" on page 1731. |

The following example sets styles by using CSS on the Legend control:

```
<?xml version="1.0"?>
<!-- charts/FormattedLegend.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Style>
     Legend {
        labelPlacement:left;
        markerHeight:30;
        markerWidth:30;
     }
  </mx:Style>

  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Expense:"Taxes", Amount:2000, Cost:321, Discount:131},
        {Expense:"Rent", Amount:1000, Cost:95, Discount:313},
        {Expense:"Bills", Amount:100, Cost:478, Discount:841}
     ]);
  ]]></mx:Script>

  <mx:Panel title="Bar Chart with Legend">
     <mx:BarChart id="myChart" dataProvider="{expenses}">
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Expense"
           />
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries
                xField="Amount"
                displayName="Amount (in $USD)"
           />
           <mx:BarSeries
                xField="Cost"
                displayName="Cost (in $USD)"
           />
           <mx:BarSeries
                xField="Discount"
                displayName="Discount (in $USD)"
           />
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

You can also change the appearance of the lines on the LegendItem object's marker. You do this with the `stroke` property or the `legendMarkerRenderer` property. For more information, see "Using strokes" on page 1731.

You can place a Legend control anywhere in your application, as long as the control has access to the scope of the chart's data. You can place the Legend control in your application without a container, inside the same container as the chart, or in its own container, such as a Panel container. The latter technique gives the Legend control a border and title bar, and lets you use the `title` attribute of the Panel to create a title, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/LegendInPanel.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Script>
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Expense:"Taxes", Amount:2000, Cost:321, Discount:131},
        {Expense:"Rent", Amount:1000, Cost:95, Discount:313},
        {Expense:"Bills", Amount:100, Cost:478, Discount:841}
     ]);
  </mx:Script>

  <mx:Panel title="Bar Chart with Legend in Panel">
     <mx:BarChart id="myChart" dataProvider="{expenses}">
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Expense"
           />
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries
                xField="Amount"
                displayName="Amount (in $USD)"
           />
           <mx:BarSeries
                xField="Cost"
                displayName="Cost (in $USD)"
           />
           <mx:BarSeries
                xField="Discount"
                displayName="Discount (in $USD)"
           />
        </mx:series>
     </mx:BarChart>
     <mx:Panel title="Legend">
        <mx:Legend dataProvider="{myChart}"/>
```

```
      </mx:Panel>
   </mx:Panel>

</mx:Application>
```

## Setting the direction of legends

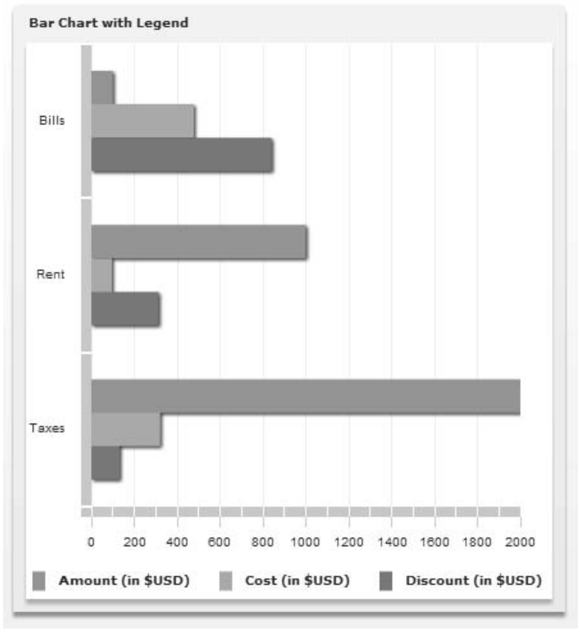
The `direction` property is a commonly used property that is inherited from the Tile container. This property of the `<mx:Legend>` tag causes the LegendItem objects to line up horizontally or vertically. The default value of `direction` is `vertical`; when you use this value, Flex stacks the LegendItem objects one on top of the other.

The following example sets the `direction` property to `horizontal`:

```
<?xml version="1.0"?>
<!-- charts/HorizontalLegend.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
backgroundColor="white">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Expense:"Taxes", Amount:2000, Cost:321, Discount:131},
        {Expense:"Rent", Amount:1000, Cost:95, Discount:313},
        {Expense:"Bills", Amount:100, Cost:478, Discount:841}
     ]);
  ]]></mx:Script>

  <mx:Panel title="Bar Chart with Legend">
     <mx:BarChart id="myChart" dataProvider="{expenses}">
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Expense"
           />
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries
                xField="Amount"
                displayName="Amount (in $USD)"
           />
           <mx:BarSeries
                xField="Cost"
                displayName="Cost (in $USD)"
           />
           <mx:BarSeries
                xField="Discount"
                displayName="Discount (in $USD)"
           />
        </mx:series>
```

```
      </mx:BarChart>
      <mx:Legend
         dataProvider="{myChart}"
         direction="horizontal"
      />
   </mx:Panel>
</mx:Application>
```

The following example shows the Legend with the `direction` property set to `horizontal`:



## Formatting the legend markers

You can define the appearance of the legend markers by using a programmatic renderer class. Flex includes several default renderer classes that you can use for legend markers.

You can change the renderer of the LegendItem object from the default to one of the ChartItem renderers by using the series' `legendMarkerRenderer` style property. This property specifies the class to use when rendering the marker in all associated legends.

The following example sets the legend marker of all three series to the DiamondItemRenderer class:

```
<?xml version="1.0"?>
<!-- charts/CustomLegendRenderer.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"January", Profit:2000, Expenses:1500, Amount:450},
        {Month:"February", Profit:1000, Expenses:200, Amount:600},
        {Month:"March", Profit:1500, Expenses:500, Amount:300},
        {Month:"April", Profit:500, Expenses:300, Amount:500},
        {Month:"May", Profit:1000, Expenses:450, Amount:250},
        {Month:"June", Profit:2000, Expenses:500, Amount:700}
     ]);

     [Bindable]
     [Embed(source="../assets/bird.gif")]
     public var myBird:Class;

     [Bindable]
     public var myBirdFactory:ClassFactory =
        new ClassFactory(myBird);

  ]]></mx:Script>
  <mx:Panel title="Plot Chart">
     <mx:PlotChart id="myChart" dataProvider="{expenses}"
     showDataTips="true">
        <mx:series>
           <!--
                Each series uses the default renderer for
                the ChartItems, but uses the same renderer
                for legend markers.
           -->

           <mx:PlotSeries
                xField="Expenses"
                yField="Profit"
                displayName="Plot 1"
                legendMarkerRenderer=
                "mx.charts.renderers.DiamondItemRenderer"
           />

           <mx:PlotSeries
                xField="Amount"
                yField="Expenses"
                displayName="Plot 2"
                legendMarkerRenderer=
                "mx.charts.renderers.DiamondItemRenderer"
           />

           <mx:PlotSeries
                xField="Profit"
                yField="Amount"
                displayName="Plot 3"
                legendMarkerRenderer=
                "mx.charts.renderers.DiamondItemRenderer"
           />
        </mx:series>
     </mx:PlotChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

If you do not explicitly set the `legendMarkerRenderer` property, the property uses the default class that the series's `itemRenderer` style property specifies. Each series has a default renderer that is used if neither of these style properties is specified.

You can create your own custom legend marker class. Classes used as legend markers must implement the IFlexDisplayObject interface and, optionally, the ISimpleStyleClient and IDataRenderer interfaces.

For more information on available renderer classes, see "Skinning ChartItem objects" on page 1783.

CHAPTER 49

# Displaying Data and Labels

49

This chapter describes some techniques for displaying data and labels in your charts.

## Contents

## Working with axes

Each chart, except for a pie chart, has horizontal and vertical axes. There are two axis types: category and numeric. A *category axis* typically defines strings that represent groupings of items in the chart; for example, types of expenses (such as rent, utilities, and insurance) or names of employees. A *numeric axis* typically defines continuous data such as the amount of an expense or the productivity gains of the employee. These data define the height of a column, for example. Numeric axes include the LinearAxis, LogAxis, and DateTimeAxis.

You work with the axes objects to define their appearance, but also to define what data is displayed in the chart.

The following sections describe the CategoryAxis and NumericAxis classes.

## About the CategoryAxis class

The CategoryAxis class maps discrete categorical data (such as states, product names, or department names) to an axis and spaces them evenly along it. This axis accepts any data type that can be represented by a String.

The `dataProvider` property of the CategoryAxis object defines the data provider that contains the text for the labels. In most cases, this can be the same data provider as the chart's data provider. A CategoryAxis object used in a chart inherits its `dataProvider` property from the containing chart, so you are not required to explicitly set the `dataProvider` property on a CategoryAxis object.

The `dataProvider` property of a CategoryAxis object can contain an Array of labels or an Array of objects. If the data provider contains objects, you use the `categoryField` property to point to the field in the data provider that contains the labels for the axis, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/BasicColumn.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

If the data provider contains an Array of labels only, you do not specify the `categoryField` property.

You can customize the labels of the CategoryAxis object rather than use the axis labels in the data provider. You do this by providing a custom data provider. To provide a custom data provider, set the value of the CategoryAxis object's `dataProvider` property to a custom Array of labels, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/CategoryAxisLabels.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);

     [Bindable]
     public var months:Array = [
        {Month:"January", monthAbbrev:"Jan"},
        {Month:"February", monthAbbrev:"Feb"},
        {Month:"March", monthAbbrev:"Mar"}
     ];


  ]]></mx:Script>
  <mx:Panel title="Line Chart">
     <mx:AreaChart
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{months}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:AreaSeries
                yField="Profit"
                displayName="Profit"
           />
           <mx:AreaSeries
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:AreaChart>
  </mx:Panel>
</mx:Application>
```

For more information about chart data providers, see "Defining chart data" on page 1612.

You can also customize the labels using the `labelFunction` property of the CategoryAxis and NumericAxis classes to point to a function that refines the labels based on the existing data provider. For more information, see "Defining axis labels" on page 1822.

## About the NumericAxis class

The NumericAxis class maps a set of continuous numerical values (such as sales volume, revenue, or profit) to coordinates on the screen. You do not typically use the NumericAxis base class directly. Instead, you use the following subclasses when you define your axis:

- LinearAxis
- LogAxis
- DateTimeAxis

These classes give you significant control over how to set the appearance and values of elements along the axis such as labels and tick marks.

You can use the `parseFunction` property to specify a custom method that formats the data points in your chart. This property is supported by all subclasses of the NumericAxis class. For a detailed description of using this property with the DateTimeAxis, see "Using the parseFunction property" on page 1805.

If you want to change the values of the labels, use the `labelFunction` property of the NumericAxis class. For more information, see "Defining axis labels" on page 1822.

## About the LinearAxis subclass

The LinearAxis subclass is the simplest of the three NumericAxis subclasses. It maps numeric values evenly between minimum and maximum values along a chart axis. By default, Flex determines the minimum, maximum, and interval values from the charting data to fit all of the chart elements on the screen. You can also explicitly set specific values for these properties. The following example sets the minimum and maximum values to 10 and 100, respectively:

```
<?xml version="1.0"?>
<!-- charts/LinearAxisSample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
  <![CDATA[
     import mx.collections.ArrayCollection;

    [Bindable]
    public var stocks:ArrayCollection = new ArrayCollection([
        {date:"2005/8/4", SMITH:37.23},
        {date:"2005/8/5", SMITH:56.53},
        {date:"2005/8/6", SMITH:17.67},
        {date:"2005/8/7", SMITH:27.72},
        {date:"2005/8/8", SMITH:85.23}
     ]);
  ]]>
  </mx:Script>

  <mx:LineChart id="mychart"
    dataProvider="{stocks}"
    showDataTips="true"
    height="300"
    width="400"
  >
     <mx:verticalAxis>
        <mx:LinearAxis
            title="linear axis"
            minimum="10"
            maximum="100"
            interval="10"
        />
     </mx:verticalAxis>

     <mx:horizontalAxis>
        <mx:CategoryAxis categoryField="date"/>
     </mx:horizontalAxis>
     <mx:series>
        <mx:LineSeries
            yField="SMITH"
            displayName="SMITH close"
        />
    </mx:series>
  </mx:LineChart>
</mx:Application>
```

## About the LogAxis subclass

The LogAxis subclass is similar to the LinearAxis subclass, but it maps values to the axis logarithmically rather than linearly. You use a LogAxis object when the data in your chart has such a wide range that clusters of data points are lost to scale. LogAxis data also cannot be rendered if it is negative. For example, if you track the stock price of a successful company since 1929, it is useful to represent the data logarithmically rather than linearly so that the chart is readable.

When you use a LogAxis object, you set a multiplier that defines the values of the labels along the axis. You set the multiplier with the `interval` property. Values must be even powers of 10, and must be greater than or equal to 0. A value of 10 generates labels at 1, 10, 100, and 1000. A value of 100 generates labels at 1, 100, and 10,000. The default value of the `interval` property is 10. The LogAxis object rounds the interval to an even power of 10, if necessary.

As with the vertical and horizontal axes, you can also set the minimum and maximum values of a LogAxis object, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/LogAxisSample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
  <![CDATA[
     import mx.collections.ArrayCollection;

    [Bindable]
    public var stocks:ArrayCollection = new ArrayCollection([
        {date:"2005/8/4", SMITH:37.23, DECKER:1500},
        {date:"2005/8/5", SMITH:56.53, DECKER:1210},
        {date:"2005/8/6", SMITH:17.67, DECKER:1270},
        {date:"2005/8/7", SMITH:27.72, DECKER:1370},
        {date:"2005/8/8", SMITH:85.23, DECKER:1530}
     ]);
  ]]>
  </mx:Script>

  <mx:LineChart id="mychart"
    dataProvider="{stocks}"
    showDataTips="true"
    height="300"
    width="400"
  >
     <mx:verticalAxis>
        <mx:LogAxis title="log axis"
            interval="10"
            minimum="10"
            maximum="10000"
        />
     </mx:verticalAxis>

     <mx:horizontalAxis>
        <mx:CategoryAxis categoryField="date"/>
     </mx:horizontalAxis>
     <mx:series>
        <mx:LineSeries yField="DECKER" displayName="DECKER close"/>
        <mx:LineSeries yField="SMITH" displayName="SMITH close" />
     </mx:series>
  </mx:LineChart>
</mx:Application>
```

## About the DateTimeAxis subclass

The DateTimeAxis subclass maps time-based values to a chart axis. The DateTimeAxis subclass calculates the minimum and maximum values to align with logical date and time units (for example, the nearest hour or the nearest week). The DateTimeAxis subclass also selects a time unit for the interval so that the chart renders a reasonable number of labels.

The `dataUnits` property of the DateTimeAxis subclass specifies how Flex should interpret the Date objects. Flex determines this property by default, but you can override it. To display data in terms of days, set the `dataUnits` property to *days*, as the following example shows:

```
<mx:DateTimeAxis dataUnits="days"/>
```

Valid values for the `dataUnits` property are `milliseconds`, `seconds`, `minutes`, `hours`, `days`, `weeks`, `months`, and `years`.

When assigning appropriate label units, a DateTimeAxis object does not assign any unit smaller than the units represented by the data. If the `dataUnits` property is set to `days`, the chart does not render labels for every hour, no matter what the minimum or maximum range is. To achieve this, you must set the value explicitly.

When using the DateTimeAxis class, you can filter out units when you set the `dataUnits` property to `days`. This lets you create a chart that shows a "work week" or some other configuration that omits certain days of the week. For more information, see "Omitting days on a DateTimeAxis" on page 1810.

Some series use the value of the `dataUnits` property to affect their rendering. Specifically, most columnar series (such as Column, Bar, Candlestick, and HLOC controls) use the value of `dataUnits` to determine how wide to render their columns. If, for example, the ColumnChart control's horizontal axis has its `dataUnits` set to weeks and `dataUnits` set to days, the ColumnChart control renders each column at one-seventh the distance between labels.

### About supported types

Data points on the DateTimeAxis object support the Date, String, and Number data types.

- Date: If the value of the data point is an instance of a Date object, it already represents an absolute date-time value and needs no interpretation. To pass a Date object as a data value, use the `parseFunction` property of the DateTimeAxis subclass. The `parseFunction` property returns a Date object. For more information, see "Using the parseFunction property" on page 1805.
- String: You can use any format that the `Date.parse()` method supports. The supported formats are:

  - `Day Month DD Hours:Minutes:Seconds GMT Year` (for example, Tue Feb 1 12:00:00 GMT-0800 2005)
  - `Day Month DD YYYY Hours:Minutes:Seconds AM|PM` (for example, Tue Feb 1 2005 12:00:00 AM)
  - `Day Month DD YYYY` (for example, Tue Feb 1 2005)
  - `MM/DD/YYYY` (for example, 02/01/2005)

  You can also write custom logic that uses the `parseFunction` property of the DateTimeAxis to take any data type and return a Date. For more information, see "Using the parseFunction property" on page 1805.

- Number: If you use a number, it is assumed to be the number of milliseconds since Midnight, 1/1/1970; for example, 543387600000. To get this value on an existing Date object, use the Date object's `getTime()` method.

The following example uses a String value for the date that matches the MM/DD/YYYY pattern and specifies that the dates are displayed in units of days:

```
<?xml version="1.0"?>
<!-- charts/DateTimeAxisSample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="600"
height="600">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection
     [Bindable]
     public var aapl:ArrayCollection = new ArrayCollection([
        {date:"08/01/2005", close:42.71},
        {date:"08/02/2005", close:42.99},
        {date:"08/03/2005", close:42.65}
     ]);
  ]]></mx:Script>
  <mx:Panel title="DateTimeAxis" width="100%" height="100%">
     <mx:LineChart id="mychart"
        dataProvider="{aapl}"
        showDataTips="true"
     >
        <mx:horizontalAxis>
           <mx:DateTimeAxis dataUnits="days"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries
                yField="close"
                xField="date"
                displayName="AAPL"
           />
        </mx:series>
     </mx:LineChart>
  </mx:Panel>
</mx:Application>
```

## Using the parseFunction property

You use the `parseFunction` property of the DateTimeAxis object to specify a method that customizes the value of the data points. With this property, you specify a method that accepts a value and returns a Date object. The Date object is then used in the DateTimeAxis object of the chart. This lets you provide customizable input strings and convert them to Date objects, which Flex can then interpret for use in the DateTimeAxis.

The parsing method specified by the `parseFunction` property is called every time a value for the DateTimeAxis must be calculated. It is called each time a data point is encountered when the user interacts with the chart. Consequently, Flex might call the parsing method often, which can degrade an application's performance. Therefore, you should try to keep the amount of code in the parsing method to a minimum.

Flex passes only one parameter to the parsing method. This parameter is the value of the data point that you specified for the series. Typically, it is a String representing some form of a date. You cannot override this parameter or add additional parameters.

The following example shows a parsing method that creates a Date object from String values in the data provider that match the "YYYY, MM, DD" pattern:

```
<?xml version="1.0"?>
<!-- charts/DateTimeAxisParseFunction.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var ABC:ArrayCollection = new ArrayCollection([
        {date:"2005, 8, 1", close:42.71},
        {date:"2005, 8, 2", close:42.99},
        {date:"2005, 8, 3", close:44}
     ]);

     public function myParseFunction(s:String):Date {
        // Get an array of Strings from the
        // comma-separated String passed in.
        var a:Array = s.split(",");

        // Trace out year, month, and day values.
        trace("y:" + a[0]);
        trace("m:" + a[1]);
        trace("d:" + a[2]);

        // To create a Date object, you pass "YYYY,MM,DD",
        // where MM is zero-based, to the Date() constructor.
        var newDate:Date = new Date(a[0],a[1]-1,a[2]);
        return newDate;
     }
  ]]></mx:Script>
  <mx:LineChart id="mychart"
    dataProvider="{ABC}"
    showDataTips="true"
  >
     <mx:horizontalAxis>
        <mx:DateTimeAxis
            dataUnits="days"
            parseFunction="myParseFunction"
        />
     </mx:horizontalAxis>
     <mx:series>
        <mx:LineSeries
            yField="close"
            xField="date"
            displayName="ABC"
        />
     </mx:series>
  </mx:LineChart>
</mx:Application>
```

## Formatting DateTimeAxis labels

When assigning the units to display along the axis, the DateTimeAxis object uses the largest unit allowed to render a reasonable number of labels. The following table describes the default label format and the minimum range for each unit type:

| Unit | Label format | Minimum range |
|---|---|---|
| Years | YYYY | If the minimum and maximum values span at least 2 years. |
| Months | MM/YY | Spans at least 2 months. |
| Weeks | DD/MM/YY | Spans at least 2 weeks. |
| Days | DD/MM/YY | Spans at least 1 day. |
| Hours | HH:MM | Spans at least 1 hour. |
| Minutes | HH:MM | Spans at least 1 minute. |
| Seconds | HH:MM:SS | Spans at least 1 second. |
| Milliseconds | HH:MM:SS:mmmm | Spans at least 1 millisecond. |

You can restrict the list of valid units for a particular chart instance to a subset that makes sense for the use case. As with a LinearAxis object, you can specify minimum, maximum, and interval values for a DateTimeAxis object.

When rounding off values, the DateTimeAxis object determines if values passed to it should be displayed in the local time zone or UTC. You can set the `displayLocalTime` property to `true` to instruct the DateTimeAxis object to treat values as local time values. The default value is `false`.

To change the values of the labels, use the `labelFunction` property of the DateTimeAxis object. This property is inherited from the NumericAxis class and is described in "Defining axis labels" on page 1822.

## Setting minimum and maximum values on a DateTimeAxis

You can define the range of values that any axis uses by setting the values of the `minimum` and `maximum` properties on that axis. For the DateTimeAxis class, however, you must use Date objects and not Numbers or Strings to define that range. To do this, you create bindable Date objects and bind the values of the `minimum` and `maximum` properties to those objects.

When creating Date objects, remember that the month parameter in the constructor is zero-based. The following example sets the minimum date for the axis to the first day of December 2006, and the maximum date for the axis to the first day of February 2007. The result is that Flex excludes the first and last data points in the ArrayCollection because those dates fall outside of the range set on the axis:

```
<?xml version="1.0"?>
<!-- charts/DateTimeAxisRange.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="600"
height="600">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     // To create a Date object, you pass "YYYY,MM,DD",
     // where MM is zero-based, to the Date() constructor.
     [Bindable]
     public var minDate:Date = new Date(2006, 11, 1);
     [Bindable]
     public var maxDate:Date = new Date(2007, 1, 1);

     [Bindable] public var myData:ArrayCollection = new
     ArrayCollection([
        {date: "11/03/2006", amt: 12345},
        {date: "12/02/2006", amt: 54331},
        {date: "1/03/2007", amt: 34343},
        {date: "2/05/2007", amt: 40299}
     ]);

  ]]></mx:Script>
  <mx:Panel title="DateTimeAxis" width="100%" height="100%">
     <mx:ColumnChart id="mychart"
        dataProvider="{myData}"
        showDataTips="true"
     >
        <mx:horizontalAxis>
           <mx:DateTimeAxis
                dataUnits="months"
                minimum="{minDate}"
                maximum="{maxDate}"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                yField="amt"
                xField="date"
                displayName="My Data"
           />
        </mx:series>
     </mx:ColumnChart>
  </mx:Panel>
</mx:Application>
```

You can also represent the range of dates in MXML by using the following syntax:

```
<mx:horizontalAxis>
  <mx:DateTimeAxis dataUnits="months">
    <mx:minimum>
      <mx:Date fullYear="2005" month="11" date="1"/>
    </mx:minimum>
    <mx:maximum>
      <mx:Date fullYear="2007" month="1" date="1"/>
    </mx:maximum>
  </mx:DateTimeAxis>
</mx:horizontalAxis>
```

## Omitting days on a DateTimeAxis

You can exclude particular days or ranges of days from a chart. This lets you create charts that show only the days of the work week, or exclude other days of the week for other reasons.

For example, if you create a LineChart control that shows a stock price over the course of an entire month, the source data typically only includes pricing data for Monday through Friday. Values for the weekend days are typically not in the data. So, the chart control extrapolates values by extending the line through the weekend days on the chart, which makes it appear as though there is data for those days. If you *disable* the weekend days, the chart control removes those days from the chart and the line only draws the days that are not disabled. There is no breakage or other indicator that there are omitted days.

To disable days of the week or ranges of days in your charts, you must set the `dataUnits` property of the DateTimeAxis to `days`. You then use the `disabledDays` or `disabledRanges` properties of the DateTimeAxis object.

The value of the `disabledDays` property of the DateTimeAxis is an Array of numbers. These numbers correspond to days of the week, with 0 being Sunday, 1 being Monday, and so on, up until 6 being Saturday.

The following example excludes Saturdays and Sundays from the chart by setting the value of the `disabledDays` property to an Array that contains 0 and 6:

```
<?xml version="1.0"?>
<!-- charts/WorkWeekAxis.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="600"
height="600">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection
     [Bindable]
     public var aapl:ArrayCollection = new ArrayCollection([
        {date:"08/01/2007", close:42},
        {date:"08/02/2007", close:43},
        {date:"08/03/2007", close:43},
        {date:"08/06/2007", close:42},
        {date:"08/07/2007", close:38},
        {date:"08/08/2007", close:37},
        {date:"08/09/2007", close:39},
        {date:"08/10/2007", close:41},
        {date:"08/13/2007", close:45},
        {date:"08/14/2007", close:47},
        {date:"08/15/2007", close:48},
        {date:"08/16/2007", close:42},
        {date:"08/17/2007", close:43},
        {date:"08/20/2007", close:45},
        {date:"08/21/2007", close:50},
        {date:"08/22/2007", close:51},
        {date:"08/23/2007", close:55},
        {date:"08/24/2007", close:51},
        {date:"08/27/2007", close:49},
        {date:"08/28/2007", close:51},
        {date:"08/29/2007", close:50},
        {date:"08/30/2007", close:49},
        {date:"08/31/2007", close:54}
     ]);

     private function myParseFunction(s:String):Date {
        var a:Array = s.split("/");
        var newDate:Date = new Date(a[2],a[0]-1,a[1]);
        return newDate;
     }

     // Create an Array that specifies which days to exclude.
     // 0 is Sunday and 6 is Saturday.
     [Bindable]
     private var offDays:Array = [0,6];

  ]]></mx:Script>
  <mx:Panel title="DateTimeAxis" width="100%" height="100%">
     <mx:LineChart id="mychart"
```

```
        dataProvider="{aapl}"
        showDataTips="true"
     >
        <mx:horizontalAxis>
           <mx:DateTimeAxis
            dataUnits="days"
            parseFunction="myParseFunction"
            disabledDays="{offDays}"
        />
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries
                yField="close"
                xField="date"
                displayName="AAPL"
           />
        </mx:series>
     </mx:LineChart>
  </mx:Panel>
</mx:Application>
```

To exclude a range of dates from the DateTimeAxis, you use the `disabledRanges` property. This property takes an Array of Date objects or date ranges. The following example excludes August 13, and then the range of days between August 27 and August 31:

```
<?xml version="1.0"?>
<!-- charts/DisabledDateRanges.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="100%"
height="100%" creationComplete="init()">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection
     [Bindable]
     public var aapl:ArrayCollection = new ArrayCollection([
        {date:"08/01/2007", close:42},
        {date:"08/02/2007", close:43},
        {date:"08/03/2007", close:43},
        {date:"08/06/2007", close:42},
        {date:"08/07/2007", close:38},
        {date:"08/08/2007", close:37},
        {date:"08/09/2007", close:39},
        {date:"08/10/2007", close:41},
        {date:"08/13/2007", close:45},
        {date:"08/14/2007", close:47},
        {date:"08/15/2007", close:48},
        {date:"08/16/2007", close:42},
        {date:"08/17/2007", close:43},
        {date:"08/20/2007", close:45},
        {date:"08/21/2007", close:50},
        {date:"08/22/2007", close:51},
        {date:"08/23/2007", close:55},
        {date:"08/24/2007", close:51},
        {date:"08/27/2007", close:49},
        {date:"08/28/2007", close:51},
        {date:"08/29/2007", close:50},
        {date:"08/30/2007", close:49},
        {date:"08/31/2007", close:54}
     ]);

     private function myParseFunction(s:String):Date {
        var a:Array = s.split("/");
        var newDate:Date = new Date(a[2],a[0]-1,a[1]);
        return newDate;
     }

     private var d1:Date, d2:Date, d3:Date;

     [Bindable]
     private var offRanges:Array = new Array ([]);

     private function init():void {
        d1 = new Date("08/13/2007");
        d2 = new Date("08/27/2007");
        d3 = new Date("08/31/2007");
        offRanges = [ d1, {rangeStart:d2, rangeEnd:d3} ];
     }

     private var series1:LineSeries;

  ]]></mx:Script>
  <mx:Panel title="DateTimeAxis" width="100%" height="100%">
     <mx:LineChart id="mychart"
        dataProvider="{aapl}"
        showDataTips="true"
     >
        <mx:horizontalAxis>
           <mx:DateTimeAxis
            dataUnits="days"
            parseFunction="myParseFunction"
            disabledRanges="{offRanges}"
        />
        </mx:horizontalAxis>

        <mx:series>
           <mx:LineSeries
                id="mySeries"
                yField="close"
                xField="date"
                displayName="AAPL"
            />
        </mx:series>
    </mx:LineChart>
  </mx:Panel>
</mx:Application>
```

The following example expands on the previous example, except it adds a DateChooser control. You can select days on the DateChooser that are then removed from the chart.

```
<?xml version="1.0"?>
<!-- charts/DisabledDateRangesWithDateChooser.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" width="100%"
height="100%" creationComplete="init()">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var aapl:ArrayCollection = new ArrayCollection([
        {date:"08/01/2007", close:42},
        {date:"08/02/2007", close:43},
        {date:"08/03/2007", close:43},
        {date:"08/06/2007", close:42},
        {date:"08/07/2007", close:38},
        {date:"08/08/2007", close:37},
        {date:"08/09/2007", close:39},
        {date:"08/10/2007", close:41},
        {date:"08/13/2007", close:45},
        {date:"08/14/2007", close:47},
        {date:"08/15/2007", close:48},
        {date:"08/16/2007", close:42},
        {date:"08/17/2007", close:43},
        {date:"08/20/2007", close:45},
        {date:"08/21/2007", close:50},
        {date:"08/22/2007", close:51},
        {date:"08/23/2007", close:55},
        {date:"08/24/2007", close:51},
        {date:"08/27/2007", close:49},
        {date:"08/28/2007", close:51},
        {date:"08/29/2007", close:50},
        {date:"08/30/2007", close:49},
        {date:"08/31/2007", close:54}
     ]);

     // Define weekend days to be removed from chart.
     [Bindable]
     private var offDays:Array = [0,6];

     [Bindable]
     private var dateChooserDisabledRanges:Array = [];

     private function init():void {
        // Limit selectable range to August of 2007 on DateChooser.
        dateChooserDisabledRanges = [
            {rangeEnd: new Date(2007, 6, 31)},
            {rangeStart: new Date(2007, 8, 1)}
        ];
```

```
         // Disable weekend days on DateChooser.
         dc1.disabledDays = [0, 6];
      }

      [Bindable]
      private var offRanges:Array = new Array([]);

      private function onDateChange(e:Event):void {
         // Get the start and end date of the range.
         var startDate:Date = e.currentTarget.selectedRanges[0].rangeStart;
         var endDate:Date = e.currentTarget.selectedRanges[0].rangeEnd;
         var d:Object = {rangeStart:startDate, rangeEnd:endDate};

         // Add object to list of ranges to disable on chart.
         offRanges.push(d);

         // Refresh the chart series with the new offRanges.
         var mySeries:Array = [];
         mySeries.push(series1);
         mychart.series = mySeries;

         // Show the current ranges.
         ta1.text = "";
         for (var i:int = 0; i < offRanges.length; i++) {
             for (var s:String in offRanges[i]) {
                 ta1.text += s + ":" + offRanges[i][s] + "\n";
             }
         }
      }

      private function clearAllDisabledDates():void {
         offRanges = [];
         dc1.selectedDate = null;
         ta1.text = "";
      }

   ]]></mx:Script>
   <mx:Panel title="DateTimeAxis" width="100%" height="100%">
      <mx:HBox>
         <mx:LineChart id="mychart"
            dataProvider="{aapl}"
            showDataTips="true"
         >
            <mx:horizontalAxis>
               <mx:DateTimeAxis
                  id="dtAxis"
                  dataUnits="days"
                  disabledDays="{offDays}"
                  disabledRanges="{offRanges}"
               />
```

```
          </mx:horizontalAxis>

          <mx:verticalAxis>
              <mx:LinearAxis minimum="30" maximum="60"/>
          </mx:verticalAxis>

          <mx:series>
             <mx:LineSeries
                  id="series1"
                  yField="close"
                  xField="date"
                  displayName="AAPL"
             />
          </mx:series>
       </mx:LineChart>
       <mx:DateChooser id="dc1"
           showToday="false"
           click="onDateChange(event)"
           displayedMonth="7"
           displayedYear="2007"
           disabledRanges="{dateChooserDisabledRanges}"
       />
    </mx:HBox>
    <mx:Button id="b1" label="Refresh" click="clearAllDisabledDates()"/>
    <mx:TextArea id="ta1" width="600" height="400"/>
  </mx:Panel>
</mx:Application>
```

## Adding axis titles

Each axis in a chart control can include a title that describes the purpose of the axis to the users. Flex does not add titles to the chart's axes unless you explicitly set them. To add titles to the axes of a chart, you use the `title` property of the axis object. This is CategoryAxis or one of the NumericAxis subclasses such as DateTimeAxis, LinearAxis, or LogAxis. To set a style for the axis title, use the `axisTitleStyleName` property of the chart control.

The following example sets the titles of the horizontal and vertical axes (in MXML and ActionScript), and applies the styles to those titles:

```
<?xml version="1.0"?>
<!-- charts/AxisTitles.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="setTitles()">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);

     private function setTitles():void {
        la1.title="Dollars";
     }

  ]]></mx:Script>

  <mx:Style>
     .myStyle {
        fontFamily:Verdana;
        fontSize:12;
        color:#4691E1;
        fontWeight:bold;
        fontStyle:italic;
     }
  </mx:Style>

  <mx:Panel>
     <mx:ColumnChart id="myChart"
        axisTitleStyleName="myStyle"
        dataProvider="{expenses}"
     >
        <mx:verticalAxis>
           <mx:LinearAxis id="la1"/>
        </mx:verticalAxis>
        <mx:horizontalAxis>
```

```
            <mx:CategoryAxis title="FY 2006"
                 categoryField="Month"
            />
         </mx:horizontalAxis>
         <mx:series>
            <mx:ColumnSeries
                 xField="Month"
                 yField="Profit"
                 displayName="Profit"
            />
            <mx:ColumnSeries
                 xField="Month"
                 yField="Expenses"
                 displayName="Expenses"
            />
         </mx:series>
      </mx:ColumnChart>
      <mx:Legend dataProvider="{myChart}"/>
   </mx:Panel>
</mx:Application>
```

You can also use embedded fonts for your axis titles. The following example embeds the font and sets the style for the vertical axis title:

```
<?xml version="1.0"?>
<!-- charts/AxisTitleEmbedFont.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
  import mx.collections.ArrayCollection;
  [Bindable]
  public var expenses:ArrayCollection = new ArrayCollection([
     {Month:"Jan", Profit:2000, Expenses:1500},
     {Month:"Feb", Profit:1000, Expenses:200},
     {Month:"Mar", Profit:1500, Expenses:500}
  ]);
  ]]></mx:Script>

  <mx:Style>
     @font-face{
        src:url("../assets/MyriadWebPro.ttf");
        fontFamily:myMyriad;
     }

     .myEmbeddedStyle {
        fontFamily:myMyriad;
        fontSize:20;
     }
  </mx:Style>

  <mx:Panel>
     <mx:ColumnChart id="column"
```

```
        dataProvider="{expenses}"
        axisTitleStyleName="myEmbeddedStyle"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
                title="FY 2006"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{column}"/>
  </mx:Panel>
</mx:Application>
```

For information on embedding fonts, see “Using embedded fonts” on page 838.

You can take advantage of the fact that the chart applies the `axisTitleStyleName` property without explicitly specifying it, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/CSSAxisTitle.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
  import mx.collections.ArrayCollection;
  [Bindable]
  public var expenses:ArrayCollection = new ArrayCollection([
     {Month:"Jan", Profit:2000, Expenses:1500},
     {Month:"Feb", Profit:1000, Expenses:200},
     {Month:"Mar", Profit:1500, Expenses:500}
  ]);
  ]]></mx:Script>

  <mx:Style>
     .axisTitles {
        color:red;
        fontWeight:bold;
        fontFamily:Arial;
        fontSize:20;
     }

     ColumnChart {
        axisTitleStyleName:axisTitles;
     }
  </mx:Style>

  <mx:Panel>
     <mx:ColumnChart id="column" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month" title="FY 2006"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
```

```
    <mx:Legend dataProvider="{column}"/>
  </mx:Panel>
</mx:Application>
```

You can also apply the title style to the axis, as the following example shows:

```
<mx:CategoryAxis title="State" styleName="myEmbeddedStyle"/>
```

## Defining axis labels

You define the values of axis labels on the horizontal axis or vertical axis. You can customize these values by using the available data in the series or you can disable these values altogether.

## Disabling axis labels

You can disable labels by setting the value of the `showLabels` property to `false` on the AxisRenderer object, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/DisabledAxisLabels.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
  import mx.collections.ArrayCollection;
  [Bindable]
  public var expenses:ArrayCollection = new ArrayCollection([
     {Month:"Jan", Profit:2000, Expenses:1500},
     {Month:"Feb", Profit:1000, Expenses:200},
     {Month:"Mar", Profit:1500, Expenses:500}
]);
  ]]></mx:Script>
  <mx:Panel>
     <mx:ColumnChart id="column" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>

        <mx:horizontalAxisRenderers>
           <mx:AxisRenderer showLabels="false"/>
        </mx:horizontalAxisRenderers>
        <mx:verticalAxisRenderers>
           <mx:AxisRenderer showLabels="false"/>
        </mx:verticalAxisRenderers>

        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{column}"/>
  </mx:Panel>
</mx:Application>
```

## Customizing axis labels

You can customize the value of axis labels by using the `labelFunction` callback function of the axis. The function specified in `labelFunction` returns a String that Flex displays as the axis label.

The callback function format for a NumericAxis object (including the DateTimeAxis, LinearAxis, and LogAxis classes) is:

```
function function_name(labelValue:Object, previousLabelValue:Object,
  axis:axis_type):return_type
```

The callback function format for a CategoryAxis object is:

```
function function_name(labelValue:Object, previousLabelValue:Object,
  axis:axis_type, labelItem:Object):return_type
```

The following table describes the parameters of the callback function:

| Parameter | Description |
|---|---|
| *labelValue* | The value of the current label. |
| *previousLabelValue* | The value of the label preceding this label. If this is the first label, the value of *previousLabelValue* is `null`. |
| *axis* | The axis object, such as CategoryAxis or NumericAxis. |
| *labelItem* | A reference to the label object. This argument is only passed in for a CategoryAxis object. For NumericAxis subclasses such as LogAxis, DateTimeAxis, and LinearAxis objects, you omit this argument.<br>This object contains a name/value pair for the chart data. For example, if the data provider defines the Month, Profit, and Expenses fields, this object might look like the following:<br>`Profit:1500`<br>`Month:Mar`<br>`Expenses:500`<br>You can access the values in this object by using dot-notation for dynamic objects, as the following example shows:<br>`return "$" + labelItem.Profit;` |
| *return_type* | The type of object that the callback function returns. This can be any object type, but is most commonly a String for CategoryAxis axes, a Number for NumericAxis objects, or a Date object for DateTimeAxis objects. |

When you use the `labelFunction`, you must be sure to import the class of the axis or the entire charts package; for example:

```
import mx.charts.*;
```

The following example defines a `labelFunction` for the horizontal CategoryAxis object. In that function, Flex appends ’07 to the axis labels, and displays the labels as Jan ’07, Feb ’07, and Mar ‘07. For the vertical axis, this example specifies that it is a LinearAxis, and formats the values to include a dollar sign and a thousands separator. The return types of the label formatting functions are Strings.

```
<?xml version="1.0"?>
<!-- charts/CustomLabelFunction.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
  import mx.charts.*;
  import mx.collections.ArrayCollection;

  [Bindable]
  public var expenses:ArrayCollection = new ArrayCollection([
     {Month: "Jan", Income: 2000, Expenses: 1500},
     {Month: "Feb", Income: 1000, Expenses: 200},
     {Month: "Mar", Income: 1500, Expenses: 500}
  ]);

  // This method customizes the values of the axis labels.
  // This signature (with 4 arguments) is for a CategoryAxis.
  public function defineLabel(
    cat:Object,
    pcat:Object,
    ax:CategoryAxis,
    labelItem:Object):String
  {
        // Show contents of the labelItem:
        for (var s:String in labelItem) {
            trace(s + ":" + labelItem[s]);
        }

        // Return the customized categoryField value:
        return cat + " '07";

        // Note that if you did not specify a categoryField,
        // cat would refer to the entire object and not the
        // value of a single field. You could then access
        // fields by using cat.field_name.
  }

  // For a NumericAxis, you do not use the labelItem argument.
  // This example uses a NumberFormatter to add a thousands
  // separator.
  public function defineVerticalLabel(
    cat:Object,
    pcat:Object,
    ax:LinearAxis):String
  {
        return "$" + numForm.format(cat);
  }

  ]]></mx:Script>

  <mx:NumberFormatter id="numForm" useThousandsSeparator="true"/>

  <mx:Panel>
     <mx:ColumnChart id="column" dataProvider="{expenses}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                categoryField="Month"
                title="Expenses"
                labelFunction="defineLabel"
           />
        </mx:horizontalAxis>

        <mx:verticalAxis>
            <mx:LinearAxis
                title="Income"
                minimum="0"
                maximum="2500"
                labelFunction="defineVerticalLabel"
            />
        </mx:verticalAxis>

        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Income"
                displayName="Income"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>
  </mx:Panel>
</mx:Application>
```

In the previous example, if you use a CategoryAxis but do not specify the value of the `categoryField` property on the axis, the label format function receives an object rather than a value for the first argument. In that case, you must drill down into the object to return a formatted String.

You can also customize labels by using the `labelFunction` property of the AxisRenderer class. This lets you control the labels if you use multiple axes. The callback function format for the AxisRenderer's label function is:

```
function function_name(axisRenderer:IAxisRenderer, label:String):String
```

Because this particular callback function takes an argument of type IAxisRenderer, you must import that class when you use this function:

```
import mx.charts.chartClasses.IAxisRenderer;
```

The following example specifies the value of the `labelFunction` property for one of the vertical axis renderers. The resulting function, `CMstoInches()` converts centimeters to inches for the axis's labels.

```
<?xml version="1.0"?>
<!-- charts/CustomLabelsOnAxisRenderer.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
    <![CDATA[
        import mx.formatters.NumberFormatter;
        import mx.charts.chartClasses.IAxisRenderer;
        import mx.collections.ArrayCollection;

       private function CMstoInches(ar:IAxisRenderer, strCMs:String):String
{
            var n:NumberFormatter = new NumberFormatter();
            n.precision = 1;
            return n.format((Number(strCMs) * 0.393700787).toString());
        }

        [Bindable]
       private var SampleHeightData:ArrayCollection = new ArrayCollection([
            { Age: "Birth", height: 53},
            { Age: "3", height: 57 },
            { Age: "6", height: 64 },
            { Age: "9", height: 70 },
            { Age: "12", height: 82 },
            { Age: "15", height: 88 }
        ]);

        [Bindable]
        private var HeightData:ArrayCollection = new ArrayCollection([
           { Age: "Birth", 5: 52, 10: 53, 25:54, 50:58, 75:60, 90:62, 95:63
},
            { Age: "3", 5: 56, 10: 57, 25:58, 50:62, 75:64, 90:66, 95:67 },
            { Age: "6", 5: 62, 10: 63, 25:64, 50:68, 75:70, 90:72, 95:73 },
            { Age: "9", 5: 66, 10: 67, 25:68, 50:72, 75:74, 90:76, 95:77 },
           { Age: "12", 5: 70, 10: 71, 25:72, 50:76, 75:80, 90:82, 95:83 },
            { Age: "15", 5: 74, 10: 75, 25:76, 50:80, 75:84, 90:86, 95:87 }
        ]);
    ]]>
    </mx:Script>

    <mx:Stroke id="s1" weight="1"  />

    <mx:Panel title="Multiple Axis Example, Boys: Age - Height percentiles"
        height="100%" width="100%" layout="horizontal">
        <mx:ColumnChart id="linechart" height="100%" width="100%"
            paddingLeft="5" paddingRight="5"
            showDataTips="true" dataProvider="{HeightData}">
```

```
<mx:seriesFilters>
    <mx:Array/>
</mx:seriesFilters>

<mx:backgroundElements>
    <mx:GridLines direction="both"/>
</mx:backgroundElements>

<mx:horizontalAxis>
    <mx:CategoryAxis id="h1"
        categoryField="Age"
        title="Age in Months"
        ticksBetweenLabels="false"
    />
</mx:horizontalAxis>

<mx:verticalAxis>
    <mx:LinearAxis id="v1" title="Height" baseAtZero="false"/>
</mx:verticalAxis>

<mx:verticalAxisRenderers>
    <mx:AxisRenderer
        axis="{v1}"
        placement="right"
    />
    <mx:AxisRenderer
        axis="{v1}"
        placement="right"
        labelFunction="CMstoInches"
        highlightElements="true"
    />
</mx:verticalAxisRenderers>

<mx:horizontalAxisRenderers>
    <mx:AxisRenderer axis="{h1}" placement="bottom"/>
    <mx:AxisRenderer axis="{h1}" placement="top"/>
</mx:horizontalAxisRenderers>

<mx:series>
    <mx:LineSeries yField="5"  form="curve" displayName="5%"/>
    <mx:LineSeries yField="10" form="curve" displayName="10%"/>
    <mx:LineSeries yField="25" form="curve" displayName="25%"/>
    <mx:LineSeries yField="50" form="curve" displayName="50%"/>
    <mx:LineSeries yField="75" form="curve" displayName="75%"/>
    <mx:LineSeries yField="90" form="curve" displayName="90%"/>
    <mx:LineSeries yField="95" form="curve" displayName="95%"/>
    <mx:ColumnSeries displayName="Height of Child X"
        dataProvider="{SampleHeightData}"
        yField="height"
```

```
                fills="{[0xCC6600]}"
            />
        </mx:series>
    </mx:ColumnChart>
    <mx:Legend dataProvider="{linechart}"/>
  </mx:Panel>
</mx:Application>
```

Another way to customize the labels on an axis is to set a custom data provider for the labels. This can be done if you are using the CategoryAxis and not the NumericAxis class for the axis values. For more information, see "About the CategoryAxis class" on page 1796.

For the PieChart control, you can customize labels with the label function defined on the PieSeries class. For more information, see "Using data labels with PieChart controls" on page 1685.

## Setting ranges

Flex determines the minimum and maximum values along an axis and sets the interval based on the settings of the NumericAxis object. You can override the values that Flex calculates. By changing the range of the data displayed in the chart, you also change the range of the tick marks.

The following table describes the properties of the axis that define the ranges along the axes:

| Property | Description |
|---|---|
| `minimum` | The lowest value of the axis. |
| `maximum` | The highest value of the axis. |
| `interval` | The number of units between values along the axis. |

The following example defines the range of the y-axis:

```
<?xml version="1.0"?>
<!-- charts/LinearAxisSample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
  <![CDATA[
     import mx.collections.ArrayCollection;

    [Bindable]
    public var stocks:ArrayCollection = new ArrayCollection([
        {date:"2005/8/4", SMITH:37.23},
        {date:"2005/8/5", SMITH:56.53},
        {date:"2005/8/6", SMITH:17.67},
        {date:"2005/8/7", SMITH:27.72},
        {date:"2005/8/8", SMITH:85.23}
     ]);
  ]]>
  </mx:Script>

  <mx:LineChart id="mychart"
    dataProvider="{stocks}"
    showDataTips="true"
    height="300"
    width="400"
  >
     <mx:verticalAxis>
        <mx:LinearAxis
            title="linear axis"
            minimum="10"
            maximum="100"
            interval="10"
        />
     </mx:verticalAxis>
```

```
      <mx:horizontalAxis>
         <mx:CategoryAxis categoryField="date"/>
      </mx:horizontalAxis>
      <mx:series>
         <mx:LineSeries
             yField="SMITH"
             displayName="SMITH close"
         />
      </mx:series>
   </mx:LineChart>
</mx:Application>
```

In this example, the minimum value displayed along the y-axis is 10, the maximum value is 100, and the interval is 10. Therefore, the label text is 10, 20, 30, 40, and so on.

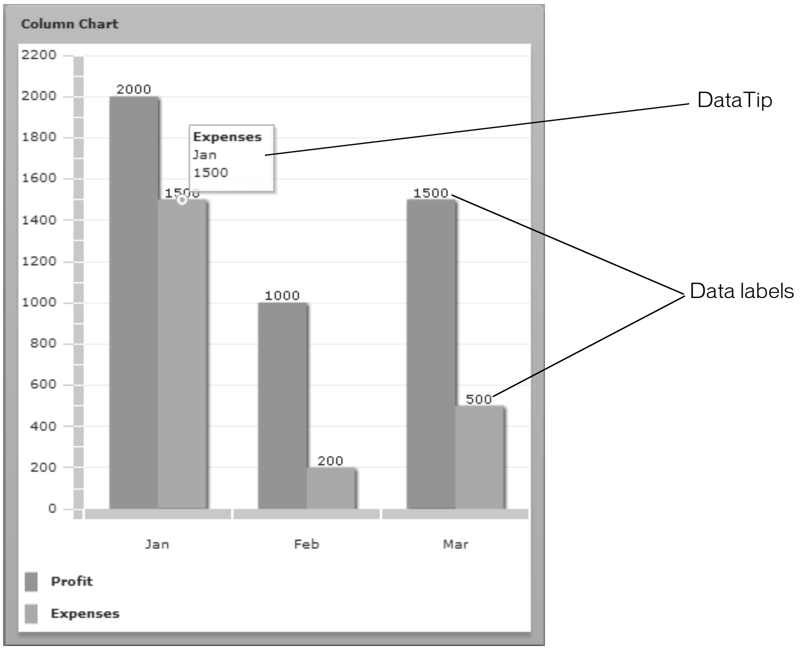
To set the minimum and maximum values on a DateTimeAxis, you must use Date objects rather than Strings or Numbers in the axis's tag. For more information, see "Setting minimum and maximum values on a DateTimeAxis" on page 1808.

For information about setting the length and location of tick marks, see "Formatting tick marks" on page 1726.

## Using data labels

Date labels show static data on the chart control. They typically appear on the chart for each chart element (such as all pie wedges or all bars) for the entire time the chart is active. They are different from DataTips in that they typically only show the simple value of a chart element in a series (for example, a column's height or a pie wedge's percentage) and do not include other information such as the series name or complex formatting. DataTips are more interactive, since they appear and disappear as the user moves the mouse over chart elements. For more information about DataTips, see "Using DataTips" on page 1844.

The following example shows a column chart with an active DataTip object and visible data labels:



The following chart series support data labels:

- BarSeries
- ColumnSeries
- PieSeries

Just like DataTips, data labels are not enabled by default. You can add data labels by setting the value of the series's `labelPosition` property to `inside` or `outside` (for BarSeries and ColumnSeries) or `inside`, `outside`, `callout`, or `insideWithCallout` (for PieSeries). For more information, see "Adding data labels" on page 1835. The default value of the series's `labelPosition` property is `none`.

The following example enables data labels on the columns by setting the value of the `labelPosition` style property to `inside`. You can click the button to change the position of the labels to the outside.

```
<?xml version="1.0"?>
<!-- charts/BasicDataLabel.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
initialize="setStartLabelLocation()">
  <mx:Script><![CDATA[
    import mx.collections.ArrayCollection;

    [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Income:2000, Expenses:1500},
        {Month:"Feb", Income:1000, Expenses:200},
        {Month:"Mar", Income:1500, Expenses:500}
     ]);

     private function setStartLabelLocation():void {
        cs1.setStyle("labelPosition", "inside");
        cs2.setStyle("labelPosition", "inside");
     }

     private function changeLabelLocation():void {
        var pos:String = cs1.getStyle("labelPosition");
        if (pos == "inside") {
            pos = "outside";
        } else {
            pos = "inside";
        }
        cs1.setStyle("labelPosition", pos);
        cs2.setStyle("labelPosition", pos);
     }

  ]]></mx:Script>

  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>

        <mx:verticalAxis>
            <mx:LinearAxis minimum="0" maximum="2500"/>
        </mx:verticalAxis>

        <mx:series>
           <mx:ColumnSeries
                id="cs1"
                xField="Month"
                yField="Income"
                displayName="Income"
           />
           <mx:ColumnSeries
                id="cs2"
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:ColumnChart>

     <mx:Button id="b1"
        label="Change Label Location"
        click="changeLabelLocation()
     />

  </mx:Panel>
</mx:Application>
```

## Adding data labels

By default, BarSeries, ColumnSeries, and PieSeries objects do not display data labels. To enabled data labels on BarSeries and ColumnSeries objects, set the value of the `labelPosition` property to `inside` or `outside`. To enabled data labels on PieSeries, set the value of the `labelPosition` property to `inside`, `outside`, `callout`, or `insideWithCallout`.

The default value of the `labelPosition` property is `none`.

Because `labelPosition` property is a style property, you can set it either inline in the series's tag or by using the `setStyle()` method, as the following example shows:

```
mySeries.setStyle("labelPosition", "outside");
```

The contents of the data label is determined by several factors, in order of precedence:

- `labelField` — Specifies a field in the data provider that sets the contents of the data label. This overrides any method specified by the `labelFunction` property.
- `labelFunction` — Specifies a method that takes several arguments and returns a String that the chart series uses for the data label's contents. For more information, see "Customizing data label values" on page 1839.

- Default — If neither the `labelField` nor the `labelFunction` are specified, then the default contents of the data label is the value that the series uses to draw the chart element. This is the `xField` value for a ColumnSeries and the `yField` value for a BarSeries. For a PieSeries, the default contents of the data labels is the value of the `field` property.

Setting the `labelPosition` property to `inside` restricts the amount of styling you can perform on the data labels. If the data labels are inside the chart elements (for example, inside the column in a ColumnSeries), their size is restricted by the available space within that element. A data label cannot be bigger than the chart element that contains it. If you try to set a data label to be larger than its underlying chart element, Flex scales and possibly truncates the contents of the data label. As a result, you should usually set the value of the `labelPosition` property to `outside`. For information about styling your data labels, see "Styling data labels" on page 1836.

For 100%, stacked, and overlaid charts, the values of the series's `labelPosition` property can only be `inside`. If you set the `labelPosition` property to `outside` for these types of bar and column series, the value is ignored and the labels are rendered as if the `labelPosition` was `inside`.

If you set the `labelPosition` property to `inside`, you cannot rotate data labels.

## Styling data labels

You can style the data labels so that their appearance fits into the style of your application. You can apply the following styles to data labels:

- `fontFamily`
- `fontSize`
- `fontStyle`
- `fontWeight`
- `labelAlign` (when `labelPosition` is `inside` only)
- `labelPosition`
- `labelSizeLimit`

You can apply these styles by using type or class selectors in CSS or by using the `setStyle()` method. You can also set these properties inline in the series's MXML tag.

The following example shows how to use CSS class selectors to set the `labelPosition` and other properties that affect the appearance of data labels in a chart:

```
<?xml version="1.0"?>
<!-- charts/StylingDataLabels.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
    import mx.collections.ArrayCollection;

    [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Income:2000, Expenses:1500},
        {Month:"Feb", Income:1000, Expenses:200},
        {Month:"Mar", Income:1500, Expenses:500}
     ]);

    /*
    Style properties on series that affect data labels:
        fontFamily; fontSize; fontStyle; fontWeight;
        labelPosition; labelRotation; labelSizeLimit
    */
  ]]></mx:Script>

  <mx:Style>
    .incomeSeries {
        fontSize:9;
        fontWeight:bold;
        labelPosition:inside;
        labelAlign:top;
    }

    .expensesSeries {
        fontSize:8;
        labelPosition:inside;
        labelAlign:middle;
    }

  </mx:Style>

  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:verticalAxis>
            <mx:LinearAxis minimum="0" maximum="2500"/>
        </mx:verticalAxis>

        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Income"
                displayName="Income"
                styleName="incomeSeries"
           />
           <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
                styleName="expensesSeries"
           />
        </mx:series>
     </mx:ColumnChart>
  </mx:Panel>
</mx:Application>
```

Labels are scaled and possibly truncated if they are too big for the area. Labels will never overlap each other or other chart elements.

For PieSeries objects, you can also specify the gap and stroke of callout lines that associate a data label with a particular wedge in the pie. For more information, see "Using data labels with PieChart controls" on page 1685.

## Aligning data labels

You can align data labels so that they are positioned, either vertically or horizontally, relative to the underlying chart element. For example, you can center a data label over a column in a ColumnChart control, or align it to the left of all bars in a BarChart control.

To align data labels you use the series' `labelAlign` style property. For ColumnSeries objects, valid values are `middle`, `top`, and `bottom`. For BarSeries objects, valid values are `left`, `center`, and `right`. The defaults are `middle` and `center`, respectively.

You can only set the labelAlign style is the `labelPosition` property is set to `inside`. Label alignment is ignored if the `labelPosition` is `outside`.

You cannot change the alignment of data labels on a PieSeries object.

## Customizing data label values

You can customize the value of your data labels. For example, you can prepend a dollar sign ($) or apply numeric formatters to the data labels to make your chart more readable. You can change the value of the data label altogther, if you want.

To customize the contents of a data label, you use the `labelFunction` property of the series to specify a callback function. The function specified in `labelFunction` returns a String that Flex displays as the data label. Flex calls this callback function for each chart element in the series when the data labels are first rendered, and calls this method any time the labels are rendered again (for example, if the data changes).

The exact signature of the custom label callback function depends on the series that you are using it with. For BarSeries and ColumnSeries objects, the function takes two arguments (see "Customizing data labels for ColumnSeries and BarSeries objects" on page 1839); for PieSeries objects, the function takes four arguments (see "Customizing data labels for PieSeries objects" on page 1841).

### Customizing data labels for ColumnSeries and BarSeries objects

For a ColumnSeries or BarSeries object, the signature for the custom label callback function is:

```
function_name (element:ChartItem, series:Series):String { }
```

The following table describes the parameters of the `labelFunction` callback function for a ColumnSeries or BarSeries object:

| Parameter | Description |
|---|---|
| `element` | A reference to the ChartItem that this data label applies to. The ChartItem represents a single data point in a series. |
| `series` | A reference to the series that this data label is used on. |

When you customize the value of the data label, you typically get a reference to the series item. You do this by casting the element argument to a specific chart item type (BarSeriesItem or ColumnSeriesItem). You then point to either the `yNumber` or `xNumber` property to get the underlying data.

You can also get a reference to the current series by casting the series argument to the specific series type (ColumnSeries or BarSeries). This lets you access properties such as the `yField` or `xField`.

The following example creates data labels for each of the columns in the ColumnChart control:

```
<?xml version="1.0"?>
<!-- charts/DataLabelFunctionColumn.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[

    import mx.charts.ChartItem;
    import mx.charts.chartClasses.Series;
    import mx.charts.series.items.ColumnSeriesItem;
    import mx.collections.ArrayCollection;

    [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Income:2000, Expenses:1500},
        {Month:"Feb", Income:1000, Expenses:200},
        {Month:"Mar", Income:1500, Expenses:500}
     ]);

     private function setCustomLabel(element:ChartItem,
series:Series):String {
        // Get a refereence to the current data element.
        var data:ColumnSeriesItem = ColumnSeriesItem(element);

        // Get a reference to the current series.
        var currentSeries:ColumnSeries = ColumnSeries(series);

        // Create a return String and format the number.
     return currentSeries.yField + ":" + " $" + nf1.format(data.yNumber);
     }

  ]]></mx:Script>

  <mx:NumberFormatter id="nf1" useThousandsSeparator="true"/>

  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>

        <mx:verticalAxis>
            <mx:LinearAxis minimum="0" maximum="3000"/>
        </mx:verticalAxis>

        <mx:series>
           <mx:ColumnSeries
                labelPosition="outside"
                xField="Month"
                yField="Income"
                displayName="Income"
                labelFunction="setCustomLabel"
           />
           <mx:ColumnSeries
                labelPosition="outside"
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
                labelFunction="setCustomLabel"
           />
        </mx:series>
     </mx:ColumnChart>
  </mx:Panel>
</mx:Application>
```

You can also access the specific field in the data provider that provides the data. For example, in the previous example, you could get the value of the data item (in this case, the value of the Income field) by using code similar to the following:

```
var data:ColumnSeriesItem = ColumnSeriesItem(element);
var s:String = data.item.Income;
return s;
```

This is not a recommended practice, however, because it makes the callback function less reusable and can force you to use additional code to detect which column you are providing data labels for.

## Customizing data labels for PieSeries objects

For a PieSeries object, the signature for the custom label callback function is:

```
function_name (data:Object, field:String, index:Number,
  percentValue:Number):String { }
```

The following table describes the parameters of the `labelFunction` callback function for a PieSeries:

| Parameter | Description |
|---|---|
| `data` | A reference to the data point that chart element represents; type Object. |
| `field` | The field name from the data provider; type String. |

| Parameter | Description |
|---|---|
| `index` | The number of the data point in the data provider; type Number. |
| `percentValue` | The size of the pie wedge relative to the pie. If the pie wedge is a quarter of the size of the pie, this value is 25; type Number. |

The following example generates data labels for a PieSeries object that include data and formatting. It defines the `display()` method as the `labelFunction` callback function to handle formatting of the label text.

```
<?xml version="1.0"?>
<!-- charts/PieLabelFunction.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.formatters.*;
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Expense:"Taxes", Amount:1900000},
        {Expense:"Salaries", Amount:1350000},
        {Expense:"Building Rent", Amount:300000},
        {Expense:"Insurance", Amount:750000},
        {Expense:"Benefits", Amount:800000},
        {Expense:"Miscellaneous", Amount:900000}
     ]);
     public function display(
        data:Object,
        field:String,
        index:Number,
        percentValue:Number):String
     {
            return data.Expense + ":$" + data.Amount +
            "\n" + round(percentValue,2) + "%";
     }
     // Rounds to 2 places:
     public function round(num:Number, precision:Number):Number {
        var result:String;
        var f:NumberFormatter = new NumberFormatter();
        f.precision = precision;
        result = f.format(num);
        return Number(result);
     }
  ]]></mx:Script>
  <mx:Panel title="Expenditures for FY04">
     <mx:PieChart id="chart"
        dataProvider="{expenses}"
        showDataTips="false"
     >
        <mx:series>
           <mx:PieSeries
                labelPosition="callout"
                field="Amount"
                labelFunction="display"
           />
        </mx:series>
     </mx:PieChart>
  </mx:Panel>
</mx:Application>
```

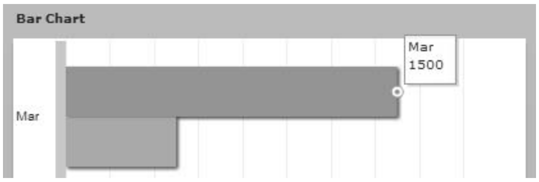
## Using DataTips

DataTips are similar to Flex ToolTip objects in that they cause a small pop-up window to appear that shows the data value for the data point under the mouse pointer. You use the `showDataTips` property of chart controls to enable DataTip objects.

NOTE

DataTip controls and data labels are not the same, although they can show the same information. Data labels are always visible regardless of the location of the user's mouse pointer. For more information about data labels, see "Using data labels" on page 1832.

The following image shows a simple DataTip:

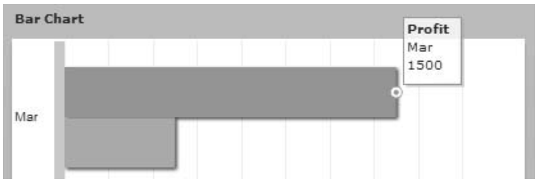


To enable DataTips, set the value of the chart control's `showDataTips` property to `true`, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/EnableDataTips.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Bar Chart">
     <mx:BarChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries
                yField="Month"
                xField="Profit"
           />
           <mx:BarSeries
                yField="Month"
                xField="Expenses"
           />
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

The format of the DataTip depends on the chart type, but typically it displays the names of the fields and the values of the data from the data provider.

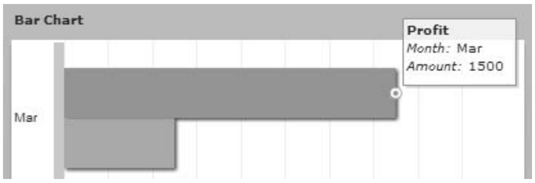
To make the information in the DataTip more understandable to users, you can define the series of your chart with names that are easily understood. Adobe Flash Player displays this name in the DataTip, as the following image shows:



The following example names the data series by using the `displayName` property of the series:

```
<?xml version="1.0"?>
<!-- charts/DataTipsDisplayName.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Bar Chart">
     <mx:BarChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries
                yField="Month"
                xField="Profit"
                displayName="Profit"
           />
           <mx:BarSeries
                yField="Month"
                xField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

You can also name the axis to display labels in the DataTip by using the `displayName` property. When the axis has a name, this name appears in the DataTip in italic font before the label data, as the following image shows:



In some cases, you add an axis solely for the purpose of adding the label to the DataTip. The following example names both axes so that both data points are labeled in the DataTip:

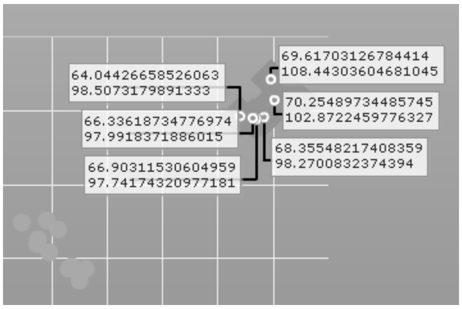
```
<?xml version="1.0"?>
<!-- charts/DataTipsAxisNames.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Bar Chart">
     <mx:BarChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
                displayName="Month"
           />
        </mx:verticalAxis>

        <mx:horizontalAxis>
           <mx:LinearAxis displayName="Amount"/>
        </mx:horizontalAxis>

        <mx:series>
           <mx:BarSeries
                yField="Month"
                xField="Profit"
                displayName="Profit"
           />
           <mx:BarSeries
                yField="Month"
                xField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

## Showing multiple DataTips

You can display more than one DataTip by using the `dataTipMode` property on the chart control. The display options are `single` and `multiple`. When `dataTipMode` is set to `multiple`, the chart displays all DataTips within range of the cursor. The following example sets the value of a ColumnChart control's `dataTipMode` property to `multiple`:

```
<?xml version="1.0"?>
<!-- charts/DataTipsMultiple.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Bar Chart">
     <mx:BarChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
        mouseSensitivity="50"
        dataTipMode="multiple"
     >
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries
                yField="Month"
                xField="Profit"
           />
           <mx:BarSeries
                yField="Month"
                xField="Expenses"
           />
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

The following example shows DataTips when the `dataTipMode` property is set to `multiple`:



The default value of `dataTipMode` depends on the chart type. Its setting is based on the likelihood that there are overlapping DataTips in that chart type. The default value of the `dataTipMode` property for the following chart types is `single`:

- BarChart
- CandlestickChart
- ColumnChart
- HLOCChart
- PieChart

The default value is `multiple` for the `dataTipMode` property for all other chart types.

To determine the size of the interactive area around a data point, you set the `mouseSensitivity` property. The `mouseSensitivity` property configures the distance, in pixels, around the data points where Flex reacts to mouse events such as `click` and `mouseOver`. With this property, you can trigger DataTips to appear when the user moves the mouse pointer *near* the data point rather than *onto* the data point. For more information, see "Changing mouse sensitivity" on page 1895.

## Customizing DataTip values

You can customize the text displayed in a DataTip by using the `dataTipFunction` callback function. When you specify a `dataTipFunction` callback function, you can access the data of the DataTip before Flex renders it and customizes the text.

The argument to the callback function is a HitData object. As a result, you must import mx.charts.HitData when using a DataTip callback function.

Flex displays whatever the callback function returns in the DataTip box. You must specify a String as the callback function's return type.

The following example defines a new callback function, `dtFunc`, that returns a formatted value for the DataTip:

```
<?xml version="1.0"?>
<!-- charts/CustomDataTips.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.charts.HitData;
     import mx.collections.ArrayCollection;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);

     public function dtFunc(hd:HitData):String {
        return hd.item.Month + ":<B>$" +
            hd.item.Profit + "</B>";
     }
  ]]></mx:Script>
  <mx:Panel title="Bar Chart">
     <mx:BarChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
        dataTipFunction="dtFunc"
     >
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
                displayName="Month"
           />
        </mx:verticalAxis>

        <mx:horizontalAxis>
           <mx:LinearAxis displayName="Amount"/>
        </mx:horizontalAxis>

        <mx:series>
           <mx:BarSeries
                yField="Month"
                xField="Profit"
                displayName="Profit"
           />
           <mx:BarSeries
                yField="Month"
                xField="Expenses"
                displayName="Expenses"
```

```
          />
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

You can also use the HitData object to get information about the series in which that data item appears. To do this, you cast the HitData object to a Series class, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/HitDataCasting.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" initialize=
"init()">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     import mx.charts.HitData;
     import mx.charts.series.ColumnSeries;

     [Bindable]
     public var dataSet:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Expenses:1500, Income:1590},
        {Month:"Feb", Expenses:1200, Income:1300},
        {Month:"Mar", Expenses:750, Income:900}
     ]);

     public var b:Boolean = true;

     public function myDataTipFunction(e:HitData):String {
        var s:String;
        s = ColumnSeries(e.element).displayName + "\n";
        s += "Profit: $" + (e.item.Income - e.item.Expenses);
        return s;
     }

  ]]></mx:Script>
  <mx:Panel title="Casting HitData Objects">
     <mx:ColumnChart id="myChart"
        dataProvider="{dataSet}"
        showDataTips="true"
        dataTipFunction="myDataTipFunction"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="Month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                yField="Expenses"
                displayName="Expenses '06"
           />
           <mx:ColumnSeries
                yField="Income"
                displayName="Income '06"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

The series item is also accessible from the HitData object in a custom DataTip function. The `chartItem` property refers to an instance of a subclass of the ChartItem class. The type depends on the series type; for example, the `chartItem` for a ColumnSeries is an instance of the ColumnSeriesItem class.

In the following example, the `yValue` of the ColumnSeriesItem represents the percentage a series takes up in a 100% chart:

```
<?xml version="1.0"?>
<!-- charts/HitDataCastingWithPercent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" initialize=
"init()">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     import mx.charts.HitData;
     import mx.charts.series.ColumnSeries;
     import mx.charts.series.items.ColumnSeriesItem;

     [Bindable]
     public var dataSet:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Income:1500, Profit:90},
        {Month:"Feb", Income:1200, Profit:100},
        {Month:"Mar", Income:750, Profit:150}
     ]);

     public var b:Boolean = true;

     public function myDataTipFunction(e:HitData):String {

        var s:String;
        s = "<B>" + ColumnSeries(e.element).displayName + "</B>\n";

        s += "<I>Income:</I> <FONT COLOR='#339966'>$" +
            e.item.Income + "</FONT>\n";
        s += "<I>Expenses:</I> <FONT COLOR='#FF0000'>$" +
            (e.item.Income - e.item.Profit) + "</FONT>\n";
        s += "------------------------\n";
        s += "<I>Profit:</I> $" + e.item.Profit + "\n";

        // The value of the Income will always be 100%,
        // so exclude adding that to the DataTip. Only
        // add percent when the user gets the Profit DataTip.
        var percentValue:Number =
            Number(ColumnSeriesItem(e.chartItem).yValue);
        if (percentValue < 100) {
            s += "Profit was equal to about <B>" +
                Math.round(percentValue) + "</B>% of the income.\n";
        }
        return s;

        //return e.item.Month + ":<B>$" + e.item.Profit + "</B>";
     }
  ]]></mx:Script>
  <mx:Panel title="Accessing ChartItems from HitData Objects">
     <mx:ColumnChart id="myChart"
        dataProvider="{dataSet}"
        type="100%"
        dataTipFunction="myDataTipFunction"
        showDataTips="true"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="Month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                yField="Profit"
                displayName="Profit '06"
           />
           <mx:ColumnSeries
                yField="Income"
                displayName="Income '06"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

For more information on using the HitData object of chart events, see “Using the HitData object” on page 1884.

The DataTip callback function can return simple HTML for formatted DataTips. Flex supports a small subset of HTML tags including `<FONT>`, `<B>`, `<I>`, and `<BR>`.

## Using per-item fills

You can customize the appearance of chart items in a series by using the `fillFunction` to define the fill. This function takes the chart item and its index as arguments, so that you can examine it's data values and return a fill based on whatever criteria you choose to use.

This lets you set a threshold for color values, or conditionalize the colors of your chart items. For example, if the size of a pie wedge is greater than 25%, make it red, or if a column's value is greater than 100, make it green.

You set the value of the `fillFunction` property on each series, so if you have multiple series, you can have multiple fill functions, or all series can share the same fill function.

The signature of the `fillFunction` is as follows:

```
function_name ( element:ChartItem, index:Number ) : IFill { }
```

The following table describes the arguments:

| Argument | Description |
|---|---|
| `element` | The chart item for which the fill is created. |
| `index` | The index of the chart item in the series's data provider. |

The `fillFunction` returns a Fill object (an object that implements the IFill interface). This object is typically an instance of the SolidColor class, but it can also be of type BitmapFill, LinearGradient, or RadialGradient. For information on working with gradients, see "Using gradient fills with chart controls" on page 1752.

The returned fill from a `fillFunction` takes precedence over fills that are set with traditional style methods. For example, if you set the value of the `fill` or `fills` property of a series and also specify a `fillFunction`, the fills are ignored and the fill returned by the `fillFunction` is used when rendering the chart items in the series.

The following example compares the value of the yField im the data provider when it fills each chart item. If the yField value (corresponding to the "CurrentAmount" field) is greater than $50,000, then the column is green. If it is less than $50,000, then the column is red:

```
<?xml version="1.0"?>
<!-- charts/SimpleFillFunction.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
    <![CDATA[

     import mx.graphics.IFill;
     import mx.graphics.SolidColor;
     import mx.collections.ArrayCollection;
     import mx.charts.ChartItem;
     import mx.charts.series.items.ColumnSeriesItem;

     [Bindable]
     public var sales:ArrayCollection = new ArrayCollection([
        { Name:"Reiner", CurrentAmount:69000 },
        { Name:"Klaus", CurrentAmount:38000 },
        { Name:"Alan", CurrentAmount:44000 },
        { Name:"Wolfgang", CurrentAmount:33000 },
        { Name:"Francis", CurrentAmount:20000 },
        { Name:"Klaus-Jurgen", CurrentAmount:55000 },
        { Name:"Martin", CurrentAmount:70000 },
        { Name:"Mac", CurrentAmount:35000 },
        { Name:"Friedemann", CurrentAmount:38000 },
        { Name:"Bruno", CurrentAmount:40000 }
     ]);

    private function myFillFunction(element:ChartItem, index:Number):IFill
{
        var c:SolidColor = new SolidColor(0x00CC00);

        var item:ColumnSeriesItem = ColumnSeriesItem(element);
        var sales:Number = Number(item.yValue);

        if (sales >= 50000) {
            return c;
        } else {
            // They have not met their goal.
            c.color = 0xFF0000;
        }
        return c;
    }

    ]]>
  </mx:Script>

  <mx:Panel title="Using a custom fillFunction in a Column Chart">
```

```
     <mx:ColumnChart id="myChart"
        dataProvider="{sales}"
        showDataTips="true"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                title="Sales Person"
                categoryField="Name"
           />
        </mx:horizontalAxis>
        <mx:verticalAxis>
            <mx:LinearAxis title="Sales (in $USD)"/>
        </mx:verticalAxis>

        <mx:series>
           <mx:ColumnSeries id="currSalesSeries"
                xField="Name"
                yField="CurrentAmount"
                fillFunction="myFillFunction"
                displayName="Current Sales"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend>
        <mx:LegendItem label="More than $50K" fontWeight="bold">
           <mx:fill>
            <mx:SolidColor color="0x00FF00"/>
           </mx:fill>
           <mx:stroke>
            <mx:Stroke color="0x000000" weight="1"/>
           </mx:stroke>
        </mx:LegendItem>
        <mx:LegendItem label="Less than $50K" fontWeight="bold">
           <mx:fill>
            <mx:SolidColor color="0xFF0000"/>
           </mx:fill>
           <mx:stroke>
            <mx:Stroke color="0x000000" weight="1"/>
           </mx:stroke>
         </mx:LegendItem>
     </mx:Legend>
  </mx:Panel>
</mx:Application>
```

This example defines custom entries in the Legend. If you use the `fillFunction` to define the fills of chart items, and you want a Legend control in your chart, then you must manually create the Legend object and its LegendItem objects. For more information on creating Legend and LegendItem objects, see "Using Legend controls" on page 1873.

By using the `index` argument that is passed to the fill function, you can also access other items inside the same series.

The following example sets the color of the fill to green for only the columns whose item value is greater than the previous item's value. Otherwise, it sets the color of the fill to red.

```
<?xml version="1.0"?>
<!-- charts/ComparativeFillFunction.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
    <![CDATA[

      import mx.graphics.IFill;
      import mx.graphics.SolidColor;
      import mx.collections.ArrayCollection;
      import mx.charts.ChartItem;
      import mx.charts.series.items.ColumnSeriesItem;

      [Bindable]
      public var sales:ArrayCollection = new ArrayCollection([
         { Name:"Reiner", CurrentAmount:69000 },
         { Name:"Klaus", CurrentAmount:38000 },
         { Name:"Alan", CurrentAmount:44000 },
         { Name:"Wolfgang", CurrentAmount:33000 },
         { Name:"Francis", CurrentAmount:20000 },
         { Name:"Klaus-Jurgen", CurrentAmount:55000 },
         { Name:"Martin", CurrentAmount:70000 },
         { Name:"Mac", CurrentAmount:35000 },
         { Name:"Friedemann", CurrentAmount:38000 },
         { Name:"Bruno", CurrentAmount:40000 }
      ]);

     private function myFillFunction(element:ChartItem, index:Number):IFill
{
         // Default to green.
         var c:SolidColor = new SolidColor(0x00FF00);

         var item:ColumnSeriesItem = ColumnSeriesItem(element);
         var sales:Number = Number(item.yValue);

         if (index == 0) {
             // The first column should be green, no matter the value.
             return c;
         } else {
             var prevVal:Number =
                 Number(currSalesSeries.items[index - 1].yValue);
             var curVal:Number =
                 Number(currSalesSeries.items[index].yValue);
             var diff:Number = curVal - prevVal;

             if (diff >= 0) {
                 // Current column's value is greater than the previous.
                 return c;
```

```
            } else {
                // Previous column's value is greater than the current.
                c.color = 0xFF0000;
            }
        }
        return c;
    }

    ]]>
  </mx:Script>

  <mx:Panel title="Using a custom fillFunction in a Column Chart">
     <mx:ColumnChart id="myChart"
        dataProvider="{sales}"
        showDataTips="true"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                title="Sales Person"
                categoryField="Name"
           />
        </mx:horizontalAxis>
        <mx:verticalAxis>
            <mx:LinearAxis title="Sales (in $USD)"/>
        </mx:verticalAxis>

        <mx:series>
           <mx:ColumnSeries id="currSalesSeries"
                xField="Name"
                yField="CurrentAmount"
                fillFunction="myFillFunction"
                displayName="Current Sales"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend>
        <mx:LegendItem label="More than Previous" fontWeight="bold">
           <mx:fill>
            <mx:SolidColor color="0x00FF00"/>
           </mx:fill>
           <mx:stroke>
            <mx:Stroke color="0x000000" weight="1"/>
           </mx:stroke>
        </mx:LegendItem>
        <mx:LegendItem label="Less than Previous" fontWeight="bold">
           <mx:fill>
            <mx:SolidColor color="0xFF0000"/>
           </mx:fill>
           <mx:stroke>
            <mx:Stroke color="0x000000" weight="1"/>
           </mx:stroke>
         </mx:LegendItem>
     </mx:Legend>
  </mx:Panel>
</mx:Application>
```

This example defines custom entries in the Legend because using a `fillFunction` prevents you from using an automatically generated Legend. For more information on creating Legend objects, see “Using Legend controls” on page 1873.

You can also access other series data inside a fill function so that you can perform comparisons against other series in a data provider. While you cannot reference a series from inside the fill function by using the fill function’s arguments, you can use the id property of a series to get access its data.

The following example uses chart items in more than one series to determine what color the fill for each chart item should be. In this case, it compares the value of the current amount of sales in the currentSalesSeries against the sales goal in the salesGoalSeries. If the goal is met, then the positive difference between the goal and the actual sales is green. If the goal is not yet met, then the negative difference between the goal and the actual sales is red.

```
<?xml version="1.0"?>
<!-- charts/ComplexFillFunction.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
    <![CDATA[

     import mx.graphics.IFill;
     import mx.graphics.SolidColor;
     import mx.collections.ArrayCollection;
     import mx.charts.ChartItem;
     import mx.charts.series.items.ColumnSeriesItem;

     [Bindable]
     public var sales:ArrayCollection = new ArrayCollection([
        { Name:"Reiner", SalesGoal:65000, CurrentAmount:69000 },
        { Name:"Klaus", SalesGoal:40000, CurrentAmount:38000 },
        { Name:"Alan", SalesGoal:40000, CurrentAmount:44000 },
        { Name:"Wolfgang", SalesGoal:48000, CurrentAmount:33000 },
        { Name:"Francis", SalesGoal:22000, CurrentAmount:20000 },
        { Name:"Klaus-Jurgen", SalesGoal:50000, CurrentAmount:55000 },
        { Name:"Martin", SalesGoal:44000, CurrentAmount:70000 },
        { Name:"Mac", SalesGoal:40000, CurrentAmount:35000 },
        { Name:"Friedemann", SalesGoal:38000, CurrentAmount:38000 },
        { Name:"Bruno", SalesGoal:42000, CurrentAmount:40000 }
     ]);

    private function myFillFunction(element:ChartItem, index:Number):IFill
{
        var item:ColumnSeriesItem = ColumnSeriesItem(element);

        trace("-----------------------------------------------");
        trace("Item index: " + index);
        trace("Sales Person: " + item.xValue);
        trace("Current Amount: " + currSalesSeries.items[index].yValue);
        trace("Sales Goal: " + item.yValue);

        // Set default color and alpha.
        var c:SolidColor = new SolidColor(0x00CC00);

        // Use the yNumber properties rather than the yValue properties
        // because the conversion to a Number is already done for
        // you. As a result, you do not have to cast them to a Number,
        // which would be less efficient.
        var goal:Number = item.yNumber;
        var currentAmount:Number = currSalesSeries.items[index].yNumber;

        // Determine if the goal was met or not.
        var diff:Number = currentAmount - goal;

        if (diff >= 0) {
            // Sales person met their goal.
            return c;
        } else if (diff < 0) {
            // Sales person did not meet their goal.
            c.color = 0xFF0000;
            c.alpha = .2;
        }
        return c;
    }

    ]]>
  </mx:Script>

  <mx:Panel title="Using a custom fillFunction in a Column Chart">
     <mx:ColumnChart id="myChart"
        dataProvider="{sales}"
        type="overlaid"
        showDataTips="true"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                title="Sales Person"
                categoryField="Name"
           />
        </mx:horizontalAxis>
        <mx:verticalAxis>
            <mx:LinearAxis title="Sales (in $USD)"/>
        </mx:verticalAxis>

        <mx:series>
           <mx:ColumnSeries id="currSalesSeries"
                xField="Name"
                yField="CurrentAmount"
                displayName="Current Sales"
            >
                <mx:fill>
                    <mx:SolidColor color="0x00FF00"/>
                </mx:fill>
           </mx:ColumnSeries>
           <mx:ColumnSeries id="salesGoalSeries"
                xField="Name"
                yField="SalesGoal"
                fillFunction="myFillFunction"
                displayName="Sales Goal"
           >
           </mx:ColumnSeries>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend>
        <mx:LegendItem label="Goal" fontWeight="bold">
           <mx:fill>
            <mx:SolidColor color="0x00CC00"/>
           </mx:fill>
           <mx:stroke>
            <mx:Stroke color="0x000000" weight="1"/>
           </mx:stroke>
        </mx:LegendItem>
        <mx:LegendItem label="Exceeded Goal" fontWeight="bold">
           <mx:fill>
            <mx:SolidColor color="0x00FF00"/>
           </mx:fill>
           <mx:stroke>
            <mx:Stroke color="0x000000" weight="1"/>
           </mx:stroke>
         </mx:LegendItem>
        <mx:LegendItem label="Missed Goal" fontWeight="bold">
           <mx:fill>
            <mx:SolidColor color="0xFF0000" alpha=".2"/>
           </mx:fill>
           <mx:stroke>
            <mx:Stroke color="0x000000" weight="1"/>
           </mx:stroke>
         </mx:LegendItem>
     </mx:Legend>
  </mx:Panel>
</mx:Application>
```

This chart uses overlaid columns, and sets the value of the top-most column's `alpha` property to .2 if it the sales goal was not met. By using an alpha property, you can view the column underneath the current column. This lets you show the difference without drawing stacked columns for only the items that have missed sales goals.

## Using the minField property

Some series types let you specify a minimum value for the elements drawn on the screen. For example, in a ColumnChart control, you can specify the base value of the column. To specify a base value (or minimum) for the column, set the value of the series object's `minField` property to the data provider field.

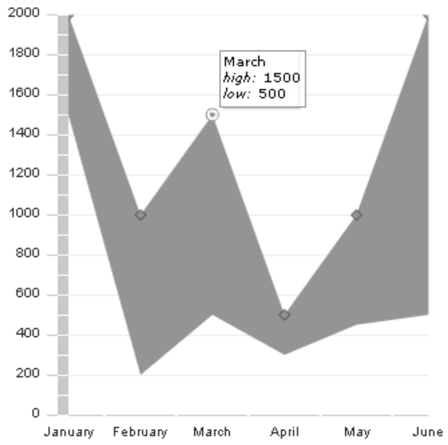
You can specify the `minField` property for the following chart series types:

- AreaSeries
- BarSeries
- ColumnSeries

Setting a value for the `minField` property creates two values on the axis for each data point in an area; for example:

```
<?xml version="1.0"?>
<!-- charts/MinField.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Area Chart">
     <mx:AreaChart
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:AreaSeries
                yField="Profit"
                minField="Expenses"
                displayName="Profit"
            />
        </mx:series>
     </mx:AreaChart>
  </mx:Panel>
</mx:Application>
```

The resulting DataTip labels the current value "high" and the `minField` value "low." The following example shows an AreaChart that defines the base of each column:



For an example of using the `minField` property to create a waterfall or cascading ColumnChart control, see "Using column charts" on page 1662.

You can also use the `baseAtZero` property to determine whether or not the axis starts at zero. Set this property to `true` to start the axis at zero; set it to `false` to let Flex determine a reasonable starting value relative to the values along the axis.

## Stacking charts

When you chart multiple data series using the AreaChart, BarChart, and ColumnChart controls, you can control how Flex displays the series using the `type` property of the controls. The following table describes the values that the `type` property supports:

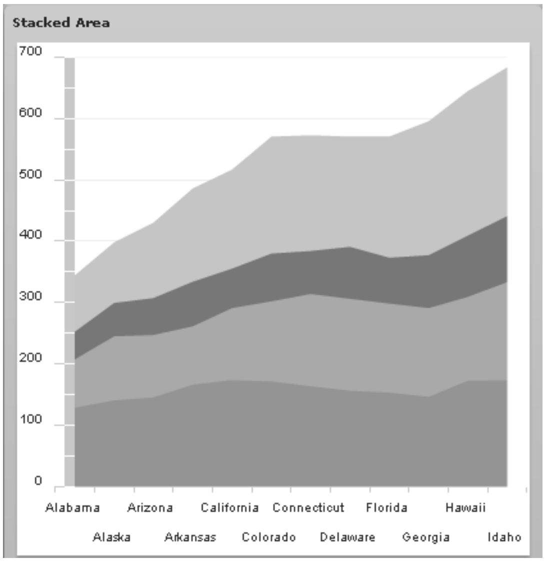
| Property | Description |
|---|---|
| `clustered` | Chart elements for each series are grouped by category. This is the default value for BarChart and ColumnChart controls. |
| `overlaid` | Chart elements for each series are rendered on top of each other, with the element corresponding to the last series on top. This is the default value for AreaChart controls. |

| Property | Description |
|---|---|
| `stacked` | Chart elements for each series are stacked on top of each other. Each element represents the cumulative value of the elements beneath it. |
| `100%` | Chart elements are stacked on top of each other, adding up to 100%. Each chart element represents the percentage that the value contributes to the sum of the values for that category. |

The following example creates an AreaChart control that has four data series, stacked on top of each other:

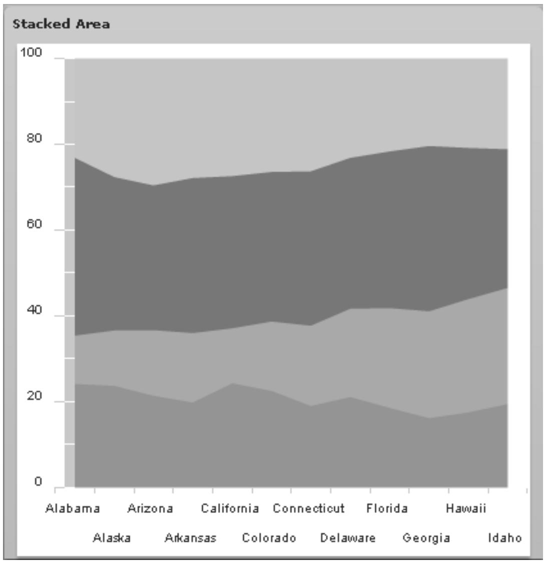
```
<?xml version="1.0"?>
<!-- charts/AreaStacked.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);
  ]]></mx:Script>
  <mx:Panel title="Stacked AreaChart">
     <mx:AreaChart
        dataProvider="{expenses}"
        showDataTips="true"
        type="stacked"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:AreaSeries
                yField="Profit"
                displayName="Profit"
           />
           <mx:AreaSeries
                yField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:AreaChart>
  </mx:Panel>
</mx:Application>
```

The following example shows a stacked AreaChart:



With an overlay, the last series appears on top, and can obscure the data series below it unless you use the `alpha` property of the fill to make it transparent. For more information, see "Using fills with chart controls" on page 1739.

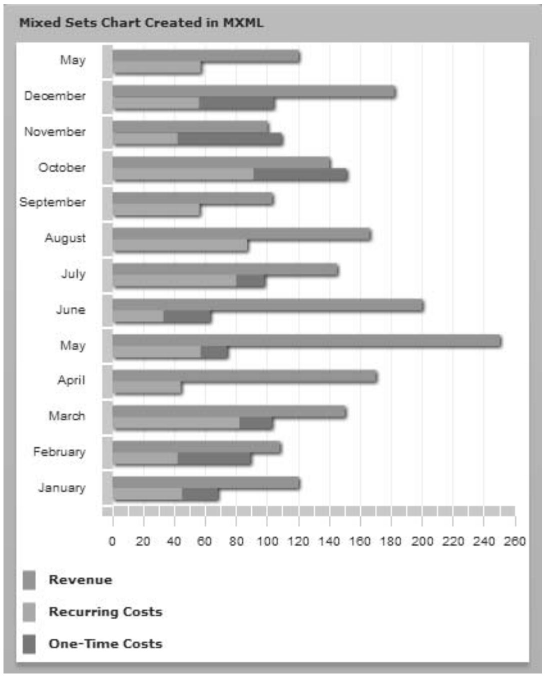
If you set the `type` property to 100%, the control draws each series stacked on top of each other, adding up to 100% of the area. Each column represents the percentage that the value contributes to the sum of the values for that category, as the following example shows:



You can use the ColumnSet, BarSet, and AreaSet classes to combine groups of chart series, and thereby use different types of series within the same chart. The following example uses BarSet classes to combine clustered and stacked BarSeries in a single chart. The example shows how to do this in MXML and ActionScript:

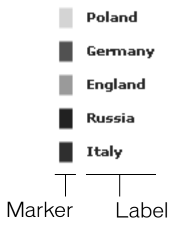
```
<?xml version="1.0"?>
<!-- charts/UsingBarSets.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
  <mx:Script><![CDATA[
     import mx.charts.Legend;
     import mx.charts.BarChart;
     import mx.charts.series.BarSet;
     import mx.charts.series.BarSeries;
     import mx.collections.ArrayCollection;

     [Bindable]
     private var yearlyData:ArrayCollection = new ArrayCollection([
        {month:"January", revenue:120, costs:45,
            overhead:102, oneTime:23},
        {month:"February", revenue:108, costs:42,
            overhead:87, oneTime:47},
        {month:"March", revenue:150, costs:82,
            overhead:32, oneTime:21},
        {month:"April", revenue:170, costs:44,
            overhead:68},
        {month:"May", revenue:250, costs:57,
            overhead:77, oneTime:17},
        {month:"June", revenue:200, costs:33,
            overhead:51, oneTime:30},
        {month:"July", revenue:145, costs:80,
            overhead:62, oneTime:18},
        {month:"August", revenue:166, costs:87,
            overhead:48},
        {month:"September", revenue:103, costs:56,
            overhead:42},
        {month:"October", revenue:140, costs:91,
            overhead:45, oneTime:60},
        {month:"November", revenue:100, costs:42,
            overhead:33, oneTime:67},
        {month:"December", revenue:182, costs:56,
            overhead:25, oneTime:48},
        {month:"May", revenue:120, costs:57,
            overhead:30}
     ]);

     private function initApp():void {
        var c:BarChart = new BarChart();
        c.dataProvider = yearlyData;

        var vAxis:CategoryAxis = new CategoryAxis();
        vAxis.categoryField = "month";
        c.verticalAxis = vAxis;

        var mySeries:Array = new Array();

        var outerSet:BarSet = new BarSet();
        outerSet.type = "clustered";
        var series1:BarSeries = new BarSeries();
        series1.xField = "revenue";
        series1.displayName = "Revenue";
        outerSet.series = [series1];

        var innerSet:BarSet = new BarSet();
        innerSet.type = "stacked";
        var series2:BarSeries = new BarSeries();
        var series3:BarSeries = new BarSeries();
        series2.xField = "costs";
        series2.displayName = "Recurring Costs";
        series3.xField = "oneTime";
        series3.displayName = "One-Time Costs";
        innerSet.series = [series2, series3];

        c.series = [outerSet, innerSet];
        var l:Legend = new Legend();
        l.dataProvider = c;

        panel2.addChild(c);
        panel2.addChild(l);
     }
  ]]></mx:Script>

  <mx:Panel title="Mixed Sets Chart Created in MXML" id="panel1">
     <mx:BarChart id="myChart" dataProvider="{yearlyData}">
        <mx:verticalAxis>
            <mx:CategoryAxis categoryField="month"/>
        </mx:verticalAxis>
        <mx:series>
            <mx:BarSet type="clustered">
                <mx:BarSeries xField="revenue"
                    displayName="Revenue"/>
                <mx:BarSet type="stacked">
                    <mx:BarSeries
                        xField="costs"
                        displayName="Recurring Costs"/>
                    <mx:BarSeries
                        xField="oneTime"
                        displayName="One-Time Costs"/>
                </mx:BarSet>
            </mx:BarSet>
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
  <mx:Panel title="Mixed Sets Chart Created in ActionScript"
    id="panel2">
  </mx:Panel>
</mx:Application>
```

The resulting chart shows two clustered series; one is a standalone series, and the other is a stacked series, as the following example shows:

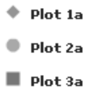


## Using Legend controls

Legend controls match the fill patterns on your chart to labels that describe the data series shown with those fill patterns. Each entry in a Legend control is known as a legend item. A legend item contains two basic parts: the marker, and the label. Legend items are of type LegendItem. The following example shows the two parts of a legend item:



In addition to matching fill patterns, legend markers also use the renderer classes of ChartItem objects, such as the points on a PlotChart control. The following example shows a Legend control for a PlotChart control with three series. Each series uses its own renderer class to draw a particular shape, and the Legend control reflects those shapes:



For information on styling Legend controls, see “Formatting Legend controls” on page 1791.

## Adding a Legend control to your chart

You use the Legend class to add a legend to your charts. The Legend control displays the label for each data series in the chart and a key that shows the chart element for the series.

The simplest way to create a Legend control is to bind a chart to it by using the `dataProvider` property, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/LegendNamedSeries.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Script>
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Expense:"Taxes", Amount:2000, Cost:321, Discount:131},
        {Expense:"Rent", Amount:1000, Cost:95, Discount:313},
        {Expense:"Bills", Amount:100, Cost:478, Discount:841}
     ]);
  </mx:Script>

  <mx:Panel title="Bar Chart with Legend">
     <mx:BarChart id="myChart" dataProvider="{expenses}">
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Expense"
           />
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries
                xField="Amount"
                displayName="Amount (in $USD)"
           />
           <mx:BarSeries
                xField="Cost"
                displayName="Cost (in $USD)"
           />
           <mx:BarSeries
                xField="Discount"
                displayName="Discount (in $USD)"
           />
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>

</mx:Application>
```

You add a Legend to a chart in ActionScript by instantiating a new object of type Legend, and then calling the container's `addChild()` method to add the Legend to the container, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/LegendInAS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="createLegend()">

  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     import mx.charts.Legend;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Expense:"Taxes", Amount:2000, Cost:321, Discount:131},
        {Expense:"Rent", Amount:1000, Cost:95, Discount:313},
        {Expense:"Bills", Amount:100, Cost:478, Discount:841}
     ]);

     private function createLegend():void {
        var myLegend:Legend = new Legend();
        myLegend.dataProvider = myChart;
        panel1.addChild(myLegend);
     }
  ]]></mx:Script>

  <mx:Panel id="panel1" title="Bar Chart with Legend (Created in
  ActionScript)">
     <mx:BarChart id="myChart" dataProvider="{expenses}">
        <mx:verticalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Expense"
           />
        </mx:verticalAxis>
        <mx:series>
           <mx:BarSeries
                xField="Amount"
                displayName="Amount (in $USD)"
           />
           <mx:BarSeries
                xField="Cost"
                displayName="Cost (in $USD)"
           />
           <mx:BarSeries
                xField="Discount"
                displayName="Discount (in $USD)"
           />
        </mx:series>
```

```
    </mx:BarChart>
  </mx:Panel>

</mx:Application>
```

The Legend control creates the legend using information from the chart control. It matches the colors and names of the LegendItem markers to the fills and labels of the chart's data series. In the previous example, Flex uses the BarSeries control's `displayName` property to define the LegendItem label.

Legend controls require that elements of the charts are named. If they are not named, the Legend markers appear, but without labels.

A Legend control for a PieChart control uses the `nameField` property of the data series to find values for the legend. The values that the `nameField` property point to must be Strings.

The following example sets the `nameField` property of a PieChart control's data series to `Expense`. Flex uses the value of the `Expense` field in the data provider in the Legend control.

```
<?xml version="1.0"?>
<!-- charts/LegendNameField.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

<mx:Script>
  import mx.collections.ArrayCollection;
  [Bindable]
  public var expenses:ArrayCollection = new ArrayCollection([
     {Expense:"Taxes", Amount:2000},
     {Expense:"Rent", Amount:1000},
     {Expense:"Food", Amount:200}
  ]);
</mx:Script>

  <mx:Panel title="Pie Chart with Legend">
     <mx:PieChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:series>
           <mx:PieSeries
                field="Amount"
                nameField="Expense"
           />
        </mx:series>
     </mx:PieChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>

</mx:Application>
```

The `nameField` property also defines the series for the DataTips and labels.

## Creating a custom Legend control

You can create a custom Legend control in MXML by defining the `<mx:Legend>` tag and populating it with `<mx:LegendItem>` tags. The following example creates a custom legend for the chart with multiple axes:

```
<?xml version="1.0"?>
<!-- charts/StyledMultipleAxes.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
      public var SMITH:ArrayCollection = new ArrayCollection([
        {date:"22-Aug-05", close:41.87},
        {date:"23-Aug-05", close:45.74},
        {date:"24-Aug-05", close:42.77},
        {date:"25-Aug-05", close:48.06},
     ]);

     [Bindable]
      public var DECKER:ArrayCollection = new ArrayCollection([
        {date:"22-Aug-05", close:157.59},
        {date:"23-Aug-05", close:160.3},
        {date:"24-Aug-05", close:150.71},
        {date:"25-Aug-05", close:156.88},
     ]);

    [Bindable]
    public var deckerColor:Number = 0x224488;

    [Bindable]
    public var smithColor:Number = 0x884422;


  ]]></mx:Script>

  <mx:Stroke id="h1Stroke"
        color="{smithColor}"
        weight="8"
        alpha=".75"
        caps="square"
  />

  <mx:Stroke id="h2Stroke"
        color="{deckerColor}"
        weight="8"
        alpha=".75"
        caps="square"
  />
```

```
<mx:Panel title="Column Chart With Multiple Axes">
   <mx:ColumnChart id="myChart" showDataTips="true">
      <mx:horizontalAxis>
         <mx:CategoryAxis id="h1" categoryField="date"/>
      </mx:horizontalAxis>

      <mx:horizontalAxisRenderers>
          <mx:AxisRenderer placement="bottom" axis="{h1}"/>
      </mx:horizontalAxisRenderers>

      <mx:verticalAxisRenderers>
          <mx:AxisRenderer placement="left" axis="{v1}">
              <mx:axisStroke>{h1Stroke}</mx:axisStroke>
          </mx:AxisRenderer>
          <mx:AxisRenderer placement="left" axis="{v2}">
              <mx:axisStroke>{h2Stroke}</mx:axisStroke>
          </mx:AxisRenderer>
      </mx:verticalAxisRenderers>

      <mx:series>
         <mx:ColumnSeries id="cs1"
              horizontalAxis="{h1}"
              dataProvider="{SMITH}"
              yField="close"
              displayName="SMITH"
          >
              <mx:fill>
                  <mx:SolidColor color="{smithColor}"/>
              </mx:fill>

              <mx:verticalAxis>
                 <mx:LinearAxis id="v1" minimum="40" maximum="50"/>
              </mx:verticalAxis>
         </mx:ColumnSeries>
         <mx:LineSeries id="cs2"
              horizontalAxis="{h1}"
              dataProvider="{DECKER}"
              yField="close"
              displayName="DECKER"
          >
              <mx:verticalAxis>
             <mx:LinearAxis id="v2" minimum="150" maximum="170"/>
              </mx:verticalAxis>

              <mx:lineStroke>
                  <mx:Stroke
                      color="{deckerColor}"
                      weight="4"
                      alpha="1"
```

```
                    />
                </mx:lineStroke>
            </mx:LineSeries>
        </mx:series>
    </mx:ColumnChart>
    <mx:Legend>
        <mx:LegendItem label="SMITH" fontWeight="bold">
            <mx:fill>
             <mx:SolidColor color="{smithColor}"/>
            </mx:fill>
            <mx:stroke>
             <mx:Stroke color="0xCCCCCC" weight="2"/>
            </mx:stroke>
        </mx:LegendItem>
        <mx:LegendItem label="DECKER" fontWeight="bold">
            <mx:fill>
             <mx:SolidColor color="{deckerColor}"/>
            </mx:fill>
            <mx:stroke>
             <mx:Stroke color="0xCCCCCC" weight="2"/>
            </mx:stroke>
         </mx:LegendItem>
    </mx:Legend>
  </mx:Panel>
</mx:Application>
```

CHAPTER 50

# Using Events and Effects in Charts

50

You can use Adobe Flex Charting to make interesting and engaging charts. Important factors to consider are how users' interactions with your applications trigger effects and events.

## Contents

## Handling user interactions with charts

Chart controls accept many kinds of user interactions, from moving the mouse over a data point to clicking or double-clicking on that data point. You can create an event handler for each of these interactions and use the Event object in that handler to access the data related to that interaction. For example, if a user clicks on a column in a ColumnChart control, you can access that column's data in the click event handler of that chart.

The chart controls support the mouse events that are inherited from the UIComponent class: `mouseMove`, `mouseOver`, `mouseUp`, `mouseDown`, and `mouseOut`. These events are of type MouseEvent. In addition, the base class for all chart controls, ChartBase, adds several chart data events. The following table describes these events:

| Chart data event | Description |
|---|---|
| `itemClick` | Broadcast when the user clicks the mouse button while over a data point. |
| `itemDoubleClick` | Broadcast when the user double-clicks the mouse button while over a data point. |
| `itemMouseDown` | Broadcast when the mouse button is down while over a data point. |
| `itemMouseMove` | Broadcast when the user moves the mouse pointer while over a data point. |
| `itemRollOut` | Broadcast when the closest data point under the mouse pointer changes. |
| `itemRollOver` | Broadcast when the user moves the mouse pointer over a new data point. |
| `itemMouseUp` | Broadcast when the user releases the mouse button while over a data point. |

Chart data events are triggered only when the user moves the mouse pointer over a data point, whereas the UIComponent events are triggered by any mouse interaction with a control.

The chart data events are of type ChartItemEvent. Because ChartItemEvent events are part of the charts package, and not part of the events package, you must import the appropriate classes in the mx.charts.events package to use a ChartItemEvent event.

The following example opens an alert when a the user clicks a data point (a column) in the chart:

```
<?xml version="1.0"?>
<!-- charts/DataPointAlert.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.controls.Alert;
     import mx.charts.events.ChartItemEvent;
     import mx.collections.ArrayCollection;

     [Bindable]
     public var dataSet:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Expenses:1500},
        {Month:"Feb", Expenses:200},
        {Month:"Mar", Expenses:500}
     ]);

     public function myHandler(e:ChartItemEvent):void {
        Alert.show("Chart data was clicked");
     }

  ]]></mx:Script>
  <mx:Panel title="Clickable Column Chart">
     <mx:ColumnChart id="myChart"
        itemClick="myHandler(event)"
        dataProvider="{dataSet}"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{dataSet}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries yField="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
  </mx:Panel>
</mx:Application>
```

## Using the HitData object

Flex generates a ChartItemEvent object for each chart data event. In addition to the standard `target` and `type` properties that all Event objects have, Flex adds the `hitData` property to the ChartItemEvent object. This property is an object instance of the HitData class. The `hitData` property contains information about the data point that is closest to the mouse pointer at the time of the mouse event.

The `item` property of the HitData object refers to the data point. You can use that property to access its value by its name, as the previous example shows. The HitData *x* and *y* properties refer to the screen coordinates of the data point.

Only data points within the radius determined by the `mouseSensitivity` property can trigger an event on that data point. For more information, see "Changing mouse sensitivity" on page 1895.

The following example uses the `itemDoubleClick` event handler to display HitData information for data points in a column chart when the user clicks a column. Because each column in the ColumnChart control is associated with a single data point value, clicking the mouse anywhere in the column displays the same information.

```
<?xml version="1.0"?>
<!-- charts/HitDataOnClick.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" initialize=
"init()">
  <mx:Script><![CDATA[
     import mx.charts.events.ChartItemEvent;
     import mx.collections.ArrayCollection;
     [Bindable]
     public var dataSet:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Expenses:1500},
        {Month:"Feb", Expenses:200},
        {Month:"Mar", Expenses:500}
     ]);
     // Define the event handler.
     public function myListener(e:ChartItemEvent):void {
        ti1.text = e.hitData.item.Expenses;
        ti2.text = e.hitData.x + "/" + e.hitData.y;
     }

     // Define event listeners when the application initializes.
     public function init():void {
        chart.addEventListener(ChartItemEvent.
            ITEM_DOUBLE_CLICK, myListener);
     }
  ]]></mx:Script>
  <mx:Panel title="Accessing HitData Object in Event Handlers">
     <mx:ColumnChart id="chart"
        dataProvider="{dataSet}" doubleClickEnabled="true"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="Month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries yField="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Form>
        <!--Define a form to display the event information.-->
        <mx:FormItem label="Expenses:">
           <mx:TextInput id="ti1"/>
        </mx:FormItem>
        <mx:FormItem label="x/y:">
           <mx:TextInput id="ti2"/>
        </mx:FormItem>
     </mx:Form>
  </mx:Panel>
</mx:Application>
```

## Getting chart elements

The HitData object accesses the chart's ChartItem objects. These objects represent data points on the screen. In addition to providing access to the data of data points, ChartItem objects provide information about the size and position of graphical elements that make up the chart. For example, you can get the *x* and *y* positions of columns in a ColumnChart.

The following example uses a semitransparent Canvas container to highlight the data point that the user clicks on with the mouse. The application also accesses the ChartItem object to get the current value to display in a ToolTip on that Canvas:

```
<?xml version="1.0"?>
<!-- charts/ChartItemObjectAccess.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="init()">

  <mx:Style>
     ToolTip {
        fontSize:24;
     }
     ColumnChart {
        gutterLeft: 54;
     }
  </mx:Style>

  <mx:Script><![CDATA[
     import mx.core.IFlexDisplayObject;
     import mx.charts.events.ChartItemEvent;
     import mx.charts.series.items.ColumnSeriesItem;
     import mx.charts.series.ColumnSeries;
     import mx.effects.Move;
     import mx.charts.HitData;
     import mx.collections.ArrayCollection;

     public var m:mx.effects.Move;
     public var hitData:mx.charts.HitData;
     public var chartItem:ColumnSeriesItem;
     public var renderer:IFlexDisplayObject;
     public var series:ColumnSeries;

     public var chartItemPoint:Point;
     public var highlightBoxPoint:Point;
     public var leftAdjust:Number;

     private function init():void {
        m = new Move(highlightBox);

        // This is used to adjust the x location of
        // highlightBox to account for the left
        // gutter width.
        leftAdjust = myChart.getStyle("gutterLeft") - 14;
     }

     [Bindable]
     private var dataSet:ArrayCollection = new ArrayCollection([
        {month:"Jan", income:12300, expense:3210},
        {month:"Feb", income:12450, expense:3100},
        {month:"Mar", income:13340, expense:3550},
        {month:"Apr", income:13489, expense:3560},
        {month:"May", income:11020, expense:4600},
        {month:"Jun", income:14030, expense:3410},
        {month:"Jul", income:15600, expense:4485},
        {month:"Aug", income:17230, expense:3892},
        {month:"Sep", income:15212, expense:3562},
        {month:"Oct", income:14980, expense:5603},
        {month:"Nov", income:15020, expense:4102},
        {month:"Dec", income:15923, expense:4789}]);

     private function overData(
        event:mx.charts.events.ChartItemEvent):void
     {

        hitData = event.hitData;
        chartItem = ColumnSeriesItem(hitData.chartItem);
        renderer = chartItem.itemRenderer;
        series = ColumnSeries(hitData.element);

        // Add 10 pixels to give it horizontal overlap.
        highlightBox.width = renderer.width * 2 + 10;

        // Add 20 pixels to give it vertical overlap.
        highlightBox.height = renderer.height + 20;

        highlightBoxPoint = new Point(highlightBox.x,
            highlightBox.y);

        // Convert the ChartItem's pixel values from local
        // (relative to the component) to global (relative
        // to the stage). This enables you to place the Canvas
        // container using the x and y values of the stage.
        chartItemPoint = myChart.localToGlobal(new
            Point(chartItem.x, chartItem.y));

        // Define the parameters of the move effect and
        // play the effect.
        m.xTo = chartItemPoint.x + leftAdjust;
        m.yTo = chartItemPoint.y;
        m.duration = 500;
        m.play();

        highlightBox.toolTip = "$" + chartItem.yValue.toString();
     }

  ]]></mx:Script>

  <mx:Panel id="p1">
     <mx:ColumnChart id="myChart"
```

```
            dataProvider="{dataSet}"
            itemClick="overData(event)"
            mouseSensitivity="0"
        >
            <mx:horizontalAxis>
                <mx:CategoryAxis categoryField="month"/>
            </mx:horizontalAxis>
            <mx:series>
                <mx:ColumnSeries
                      displayName="Expense"
                      yField="expense"
                />
                <mx:ColumnSeries
                      displayName="Income"
                      yField="income"
                />
            </mx:series>
        </mx:ColumnChart>
    </mx:Panel>

    <!-- Define the canvas control that will be used as a highlight -->
    <mx:Canvas id="highlightBox"
            y="0"
            x="0"
            backgroundColor="0xFFFF00"
            alpha=".5"
    />
</mx:Application>
```

For information about changing the appearance of ChartItem objects, see "Skinning ChartItem objects" on page 1785.

## Getting data with coordinates

When you create charts, you can use the coordinates on the screen to get the nearest data point or data points, or conversely, pass the data point and get the coordinates.

The `findDataPoints()` and `localToData()` methods take coordinates and return data. The `findDataPoints()` method returns a HitData object. For more information, see "Using the findDataPoints() method" on page 1890. The `localToData()` method returns an Array of the data. For more information, see "Using the localToData() method" on page 1892.

You can also pass the data itself and get the coordinates with the `dataToLocal()` method. For more information, see "Using the dataToLocal() method" on page 1894.

### Using the findDataPoints() method

You can use the chart control's `findDataPoints()` method to get an Array of HitData objects by passing in x and y coordinates. If the coordinates do not correspond to the location of a data point, the `findDataPoints()` method returns `null`. Otherwise, the `findDataPoints()` method returns an Array of HitData objects.

The `findDataPoints()` method has the following signature:

```
findDataPoints(x:Number, y:Number):Array
```

The following example creates a PlotChart control and records the location of the mouse pointer as the user moves the mouse over the chart. It uses the `findDataPoints()` method to get an Array of HitData objects, and then displays some of the first object's properties.

```
<?xml version="1.0"?>
<!-- charts/FindDataPoints.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.charts.HitData;
     import mx.collections.ArrayCollection;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"January", Profit:2000, Expenses:1500},
        {Month:"February", Profit:1000, Expenses:200},
        {Month:"March", Profit:1500, Expenses:500},
        {Month:"April", Profit:500, Expenses:300},
        {Month:"May", Profit:1000, Expenses:450},
        {Month:"June", Profit:2000, Expenses:500}]);

     public function handleMouseMove(e:MouseEvent):void {
        // Use coordinates to get HitData object of
        // current data point.
        var hda:Array =
            chart.findDataPoints(e.currentTarget.mouseX,
            e.currentTarget.mouseY);
        if (hda[0]) {
           ta.text = "Found data point " +
                hda[0].chartItem.index + " (x/y):" +
                Math.round(hda[0].x) + "," +
                Math.round(hda[0].y) + "\n";
           ta.text += "Expenses:" + hda[0].item.Expenses;
        } else {
           ta.text = "No data point found  (x/y):" +
                Math.round(e.currentTarget.mouseX) +
                "/" + Math.round(e.currentTarget.mouseY);
        }
     }
  ]]></mx:Script>
  <mx:Panel title="Plot Chart">
     <mx:PlotChart id="chart"
        mouseMove="handleMouseMove(event)"
        dataProvider="{expenses}"
        showDataTips="true"
        mouseSensitivity="5"
     >
        <mx:series>
           <mx:PlotSeries
                xField="Profit"
                yField="Expenses"
```

```
          />
        </mx:series>
      </mx:PlotChart>
  </mx:Panel>
  <mx:TextArea id="ta" width="300" height="50"/>
</mx:Application>
```

### Using the localToData() method

The `localToData()` method takes a Point object that represents the x and y coordinates you want to get the data for and returns an Array of data values, regardless of whether any data items are at or near that point.

The following example creates a Point object from the mouse pointer's location on the screen and displays the data values associated with that point:

```
<?xml version="1.0"?>
<!-- charts/LocalToData.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     public var p:Point;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {Month:"Feb", Profit:1000, Expenses:200},
        {Month:"Mar", Profit:1500, Expenses:500}
     ]);

     private function updateDetails(e:MouseEvent):void {
        p = new Point(myChart.mouseX,myChart.mouseY);
        mpos.text = "(" + p.x + "," + p.y + ")";

        var d:Array = myChart.localToData(p);
        dval.text = "(" + d[0] + "," + Math.floor(d[1]) + ")";

        p = myChart.dataToLocal(d[0],d[1]);
        dpos.text ="(" + Math.floor(p.x) + "," +
            Math.floor(p.y) + ")";
     }


  ]]></mx:Script>
  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart"
        dataProvider="{expenses}"
        mouseMove="updateDetails(event)"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
           <mx:ColumnSeries
                xField="Month"
                yField="Profit"
                displayName="Profit"
           />
           <mx:ColumnSeries
```

```
              xField="Month"
              yField="Expenses"
              displayName="Expenses"
          />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>


  <mx:Form width="300">
     <mx:FormItem label="Mouse Position:">
        <mx:Label id="mpos"/>
     </mx:FormItem>
     <mx:FormItem label="Data Position:">
        <mx:Label id="dpos"/>
     </mx:FormItem>
     <mx:FormItem label="DATA:">
        <mx:Label id="dval"/>
     </mx:FormItem>
  </mx:Form>

</mx:Application>
```

Individual chart types determine how coordinates are mapped, and how many values are returned in the Array. The values returned are typically numeric values.

In a chart that is based on the CartesianChart class (for example, a BarChart or ColumnChart control), the first item in the returned Array is the value of the *x*-coordinate along the horizontal axis, and the second item is the value of the *y*-coordinate along the vertical axis.

In a chart based on the PolarChart class (such as PieChart), the returned Array maps the coordinates to a set of polar coordinates—an angle around the center of the chart, and a distance from the center. Those values are mapped to data values that use the first (angular) and second (radial) axes of the chart.

## Using the dataToLocal() method

The `dataToLocal()` method converts a set of values to x and y coordinates on the screen. The values you give the method are in the “data space” of the chart; this method converts these values to coordinates. The *data space* is the collection of all possible combinations of data values that a chart can represent.

The number and meaning of arguments passed to the `dataToLocal()` method depend on the chart type. For CartesianChart controls, such as the BarChart and ColumnChart controls, the first value is used to map along the x axis, and the second value is used to map along the y axis.

For PolarChart controls, such as the PieChart control, the first value maps to the angle around the center of the chart, and the second value maps to the distance from the center of the chart along the radius.

The coordinates returned are based on 0,0 being the upper-left corner of the chart. For a ColumnChart control, for example, the height of the column is inversely related to the x coordinate that is returned.

## Changing mouse sensitivity

You use the `mouseSensitivity` property of the chart control to determine when the mouse pointer is considered to be over a data point; for example:

```
<mx:ColumnChart id="chart" dataProvider="{dataSet}" mouseSensitivity="30">
```

The current data point is the nearest data point to the mouse pointer that is less than or equal to the number of pixels that the `mouseSensitivity` property specifies.

The default value of the `mouseSensitivity` property is 3 pixels. If the mouse pointer is 4 or more pixels away from a data point, Flex does not trigger a chart data event (such as `itemRollOver` or `itemClick`). Flex still responds to events such as `mouseOver` and `click` by generating an Event object.

The following example initially sets the `mouseSensitivity` property to 20, but lets the user change this value with the HSlider control:

```
<?xml version="1.0"?>
<!-- charts/MouseSensitivity.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"January", Profit:2000, Expenses:1500, Amount:450},
        {Month:"February", Profit:1000, Expenses:200, Amount:600},
        {Month:"March", Profit:1500, Expenses:500, Amount:300},
        {Month:"April", Profit:500, Expenses:300, Amount:500},
        {Month:"May", Profit:1000, Expenses:450, Amount:250},
        {Month:"June", Profit:2000, Expenses:500, Amount:700}
     ]);

  ]]></mx:Script>
  <mx:Panel title="Mouse Sensitivity">
     <mx:PlotChart id="chart"
        dataProvider="{expenses}"
        showDataTips="true"
        mouseSensitivity="{mySlider.value}"
    >
        <mx:series>
           <mx:PlotSeries
                xField="Expenses"
                yField="Profit"
                displayName="P 1"
           />
           <mx:PlotSeries
                xField="Amount"
                yField="Expenses"
                displayName="P 2"
           />
           <mx:PlotSeries
                xField="Profit"
                yField="Amount"
                displayName="P 3"
           />
        </mx:series>
     </mx:PlotChart>

     <mx:HSlider id="mySlider"
        minimum="0"
        maximum="300"
        value="20"
        dataTipPlacement="top"
        tickColor="black"
```

```
            snapInterval="1"
            tickInterval="20"
            labels="['0','300']"
            allowTrackClick="true"
            liveDragging="true"
        />
    </mx:Panel>

</mx:Application>
```

You can use the `mouseSensitivity` property to increase the area that triggers DataTip events or emits chart-related events. If the mouse pointer is within the range of multiple data points, Flex chooses the closest data point. For DataTip events, if you have multiple DataTip controls enabled, Flex displays all DataTip controls within range. For more information, see "Showing multiple DataTips" on page 1850.

## Disabling interactivity

You can make the series in a chart ignore all mouse events by setting the `interactive` property to `false` for that series. The default value is `true`. This lets you disable mouse interactions for one series while allowing it for another.

The following example disables interactivity for events on the first and third PlotSeries objects:

```
<?xml version="1.0"?>
<!-- charts/DisableInteractivity.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"January", Profit:2000, Expenses:1500, Amount:450},
        {Month:"February", Profit:1000, Expenses:200, Amount:600},
        {Month:"March", Profit:1500, Expenses:500, Amount:300},
        {Month:"April", Profit:500, Expenses:300, Amount:500},
        {Month:"May", Profit:1000, Expenses:450, Amount:250},
        {Month:"June", Profit:2000, Expenses:500, Amount:700}
     ]);

  ]]></mx:Script>
  <mx:Panel title="Disable Interactivity">
     <mx:PlotChart id="chart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:series>
           <mx:PlotSeries
                xField="Expenses"
                yField="Profit"
                displayName="P 1"
                interactive="false"
           />
           <mx:PlotSeries
                xField="Amount"
                yField="Expenses"
                displayName="P 2"
           />
           <mx:PlotSeries
                xField="Profit"
                yField="Amount"
                displayName="P 3"
                interactive="false"
           />
        </mx:series>
     </mx:PlotChart>
  </mx:Panel>
</mx:Application>
```

Disabling the series interactivity has the following results:

- The series does not show DataTip controls.
- The series does not generate a `hitData` structure on any chart data event.

- The series does not return a `hitData` structure when you call the `findDataPoints()` method on the chart.

## Using effects with charts

Chart controls support the standard Flex effects such as Zoom and Fade. You can use these effects to make the entire chart control zoom in or fade out in your Flex applications. In addition, chart data series support the following effect classes that apply to the data in the chart:

- SeriesInterpolate
- SeriesSlide
- SeriesZoom

These effects zoom or slide the chart items inside the chart control. These classes are in the mx.charts.effects.effects package. The base class for chart effects is SeriesEffect.

All chart controls and series support the standard Flex triggers and effects that are inherited from UIComponent. These triggers include `focusInEffect`, `focusOutEffect`, `moveEffect`, `showEffect`, and `hideEffect`. The Flex Charting controls also include the `showDataEffect` and `hideDataEffect` triggers.

For information on creating complex effects, see Chapter 19, "Using Behaviors," on page 719.

### Using standard effect triggers

Chart controls support standard effects triggers, such as `showEffect` and `hideEffect`.

The following example defines a set of wipe effects that Flex executes when the user toggles the chart's visibility by using the Button control. Also, Flex fades in the chart when it is created.

```
<?xml version="1.0"?>
<!-- charts/StandardEffectTriggers.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" 
creationComplete="init()">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     import mx.effects.Fade;

     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500, Amount:450},
        {Month:"Feb", Profit:1000, Expenses:200, Amount:600},
        {Month:"Mar", Profit:1500, Expenses:500, Amount:300}
     ]);

  ]]></mx:Script>

  <!-- Define the effects -->
  <mx:Parallel id="showEffects">
     <mx:WipeRight duration="2000"/>
     <mx:Fade alphaFrom="0" alphaTo="1" duration="4000"/>
  </mx:Parallel>

  <mx:Parallel id="hideEffects">
     <mx:Fade alphaFrom="1" alphaTo="0" duration="2500"/>
     <mx:WipeLeft duration="3000"/>
  </mx:Parallel>

  <mx:Panel title="Area Chart with Effects">
     <mx:AreaChart id="myChart" 
        dataProvider="{expenses}" 
        creationCompleteEffect="showEffects" 
        hideEffect="hideEffects" 
        showEffect="showEffects"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="Month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:AreaSeries 
                yField="Profit" 
                displayName="Profit"
           />
           <mx:AreaSeries 
                yField="Expenses" 
                displayName="Expenses"
```

```
            />
        </mx:series>
     </mx:AreaChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>

  <mx:Button label="Toggle visibility"
        click="myChart.visible=!myChart.visible"
  />
</mx:Application>
```

A negative aspect of using standard effect triggers with charts is that the effects are applied to the entire chart control, and not just the data in the chart control. The result is that an effect such as Fade causes the chart's axes, gridlines, labels, and other chart elements, in addition to the chart's data, to fade in or out during the effect. To solve this, you use special charting effect triggers (see "Using charting effect triggers" on page 1901).

## Using charting effect triggers

Charts have two unique effect triggers: `hideDataEffect` and `showDataEffect`. You set these triggers on the data series for the chart. Whenever the data for a chart changes, Flex executes these triggers in succession on the chart's data. The chart control's other elements, such as gridlines, axis lines, and labels are not affected by the effect.

The `hideDataEffect` trigger defines the effect that Flex uses as it hides the current data from view. The `showDataEffect` trigger defines the effect that Flex uses as it moves the current data into its final position on the screen.

Because Flex triggers the effects associated with `hideDataEffect` and `showDataEffect` when the data changes, there is "old" data and "new" data. The effect associated with the `hideDataEffect` trigger is the "old" data that will be replaced by the new data.

The order of events is as follows:

1. Flex first uses the `hideDataEffect` trigger to invoke the effect set for each element of a chart that is about to change. These triggers execute at the same time.
2. Flex then updates the chart with its new data. Any elements (including the grid lines and axes) that do not have effects associated with them are updated immediately with their new values.
3. Flex then invokes the effect set with the `showDataEffect` trigger for each element associated with them. Flex animates the new data in the chart.

## Charting effects with data series

Three effects are unique to charting: SeriesInterpolate, SeriesSlide, and SeriesZoom. You use these effects on a data series to achieve an effect when the data in that series changes. These effects can have a great deal of visual impact. Data series do not support other Flex effects.

The following example shows how you can use the SeriesSlide effect to make the data slide in and out of a screen when the data changes. You can trigger changes to data in a chart series in many ways. Most commonly, you trigger an effect when the data provider changes for a chart control. In the following example, when the user clicks the button, the chart controls data provider changes, triggering the effect:

```
<?xml version="1.0"?>
<!-- charts/BasicSeriesSlideEffect.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var expenses1:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Income:2000, Expenses:1500},
        {Month:"Feb", Income:1000, Expenses:200},
        {Month:"Mar", Income:1500, Expenses:500}
     ]);

     [Bindable]
     public var expenses2:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Income:1200, Expenses:800},
        {Month:"Feb", Income:2500, Expenses:300},
        {Month:"Mar", Income:575, Expenses:490}
     ]);

     public function changeProvider():void {
        myChart.dataProvider=expenses2;
     }

  ]]></mx:Script>

  <!-- Define chart effects -->
  <mx:SeriesSlide
    id="slideIn"
    duration="1000"
    direction="up"
  />
  <mx:SeriesSlide
    id="slideOut"
    duration="1000"
    direction="down"
  />

  <mx:Panel title="Column Chart with Basic Series Slide Effect">
     <mx:ColumnChart id="myChart" dataProvider="{expenses1}">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses1}"
                categoryField="Month"
            />
        </mx:horizontalAxis>

        <mx:verticalAxis>
           <mx:LinearAxis minimum="0" maximum="3000"/>
        </mx:verticalAxis>

        <mx:series>
           <mx:ColumnSeries
            xField="Month"
            yField="Income"
            displayName="Income"
            showDataEffect="slideIn"
            hideDataEffect="slideOut"
           />
           <mx:ColumnSeries
            xField="Month"
            yField="Expenses"
            displayName="Expenses"
            showDataEffect="slideIn"
            hideDataEffect="slideOut"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>

  <mx:Button id="b1" click="changeProvider()"
    label="Change Data Provider"
  />

</mx:Application>
```

This example explicitly defines the minimum and maximum values of the vertical axis. If it did not, Flex would recalculate these values when the new data provider was applied. The result would be a change in the axis labels during the effect.

Changing a data provider first triggers the `hideDataEffect` effect on the original data provider, which causes that data provider to "slide out," and then triggers the `showDataEffect` effect on the new data provider, which causes that data provider to "slide in."

NOTE

If you set the data provider on the series and not the chart control, you must change it on the series and not the chart control.

Another trigger of the data effects is when a new data point is added to a series. The following example triggers the `showDataEffect` when the user clicks the button and adds a new item to the series' data provider:

```
<?xml version="1.0"?>
<!-- charts/AddItemEffect.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script><![CDATA[
        import mx.collections.ArrayCollection;

        [Bindable]
        public var items:ArrayCollection = new ArrayCollection([
            {item: 2000},
            {item: 3300},
            {item: 3000},
            {item: 2100},
            {item: 3200}
        ]);

        public function addDataItem():void {
            // Add a randomly generated value to
            // the data provider.
            var n:Number = Math.random() * 3000;
            var o:Object = {item: n};
            items.addItem(o);
        }
    ]]></mx:Script>

    <!-- Define chart effect -->
    <mx:SeriesSlide id="slideIn"
        duration="1000"
        direction="up"
    />

    <mx:Panel title="Column Chart with Series Effect">
        <mx:ColumnChart id="myChart" dataProvider="{items}">
            <mx:series>
                <mx:ColumnSeries
                    yField="item"
                    displayName="Quantity"
                    showDataEffect="slideIn"
                />
            </mx:series>
        </mx:ColumnChart>
    </mx:Panel>

    <mx:Button id="b1"
        click="addDataItem()"
        label="Add Data Item"
    />
```

```
</mx:Application>
```

For more information about changing charting data at run time, see "Changing chart data at run time" on page 1633.

The charting effects have several properties in common that traditional effects do not have. All of these properties are optional. The following table lists the common properties of the charting effects:

| Property | Description |
|---|---|
| `duration` | The amount of time, in milliseconds, that Flex takes to complete the entire effect. This property defines the speed with which the effect executes.<br>The `duration` property acts as a minimum duration. The effect can take longer based on the settings of other properties.<br>The default value is 500. |
| `elementOffset` | The amount of time, in milliseconds, that Flex delays the effect on each element in the series.<br>Set the `elementOffset` property to 0 to make all elements start at the same time and end at the same time.<br>Set the `elementOffset` property to an integer such as 30 to stagger the effect on each element by that amount of time. For example, with a slide effect, the first element slides in immediately. The next element begins 30 milliseconds later, and so on. The amount of time for the effect to execute is the same for each element, but the overall duration of the effect is longer.<br>Set the `elementOffset` property to a negative value to make the effect display the last element in the series first.<br>The default value is 20. |

| Property | Description |
|---|---|
| `minimumElementDuration` | The amount of time, in milliseconds, that an individual element should take to complete the effect.<br>Charts with a variable number of data points in the series cannot reliably create smooth effects with only the `duration` property.<br>For example, an effect with a duration of 1000 and `elementOffset` of 100 takes 900 milliseconds per element to complete if you have two elements in the series. This is because the start of each effect is offset by 100, and each effect finishes in 1000 milliseconds.<br>If there are four elements in the series, each element takes 700 milliseconds to complete (the last effect starts 300 milliseconds after the first and must be completed within 1000 milliseconds). With 10 elements, each element has only 100 milliseconds to complete the effect.<br>The value of the `minimumElementDuration` property sets a minimal duration for each element. No element of the series takes less than this amount of time (in milliseconds) to execute the effect. As a result, it is possible for an effect to take longer than a specified duration if you specify at least two of the following three properties: `duration`, `offset`, and `minimumElementDuration`.<br>The default value is 0. |
| `offset` | The amount of time, in milliseconds, that Flex delays the start of the effect. Use this property to stagger effects on multiple series.<br>The default value is 0. |

## Using the SeriesSlide effect

The SeriesSlide effect slides a data series into and out of the chart's boundaries. The SeriesSlide effect takes a `direction` property that defines the location from which the series slides. Valid values of `direction` are `left`, `right`, `up`, or `down`.

If you use the SeriesSlide effect with a `hideDataEffect` trigger, the series slides from the current position onscreen to a position off the screen, in the direction indicated by the `direction` property. If you use SeriesSlide with a `showDataEffect` trigger, the series slides from off the screen to a position on the screen, in the indicated direction.

When you use the SeriesSlide effect, the entire data series disappears from the chart when the effect begins. The data then reappears based on the nature of the effect. To keep the data on the screen at all times during the effect, you can use the SeriesInterpolate effect. For more information, see "Using the SeriesInterpolate effect" on page 1911.

The following example creates an effect called `slideDown`. Each element starts its slide 30 milliseconds after the element before it, and takes at least 20 milliseconds to complete its slide. The entire effect takes at least 1 second (1000 milliseconds) to slide the data series down. Flex invokes the effect when it clears old data from the chart and when new data appears.

```
<?xml version="1.0"?>
<!-- charts/CustomSeriesSlideEffect.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var items:ArrayCollection = new ArrayCollection([
        {item:2000},
        {item:3300},
        {item:3000},
        {item:2100},
        {item:3200}
     ]);

     public function addDataItem():void {
        // Add a randomly generated value to the data provider
        var n:Number = Math.random() * 3000;
        var o:Object = {item:n};
        items.addItem(o);
     }
  ]]></mx:Script>

  <!-- Define chart effect -->
  <mx:SeriesSlide duration="1000"
    direction="down"
    minimumElementDuration="20"
    elementOffset="30"
    id="slideDown"
  />

  <mx:Panel title="Column Chart with Custom Series Slide Effect">
     <mx:ColumnChart id="myChart" dataProvider="{items}">
        <mx:series>
           <mx:ColumnSeries
            yField="item"
            displayName="Quantity"
            showDataEffect="slideDown"
            hideDataEffect="slideDown"
           />
        </mx:series>
     </mx:ColumnChart>
  </mx:Panel>
  <mx:Button id="b1"
    click="addDataItem()"
    label="Add Data Item"
  />
</mx:Application>
```

## Using the SeriesZoom effect

The SeriesZoom effect implodes and explodes chart data into and out of the focal point that you specify. As with the SeriesSlide effect, whether the effect is zooming to or from this point depends on whether it's used with a `showDataEffect` or `hideDataEffect` trigger.

The SeriesZoom effect can take several properties that define how the effect acts. The following table describes these properties:

| Property | Description |
|---|---|
| `horizontalFocus` `verticalFocus` | Defines the location of the focal point of the zoom.<br>You combine the `horizontalFocus` and `verticalFocus` properties to define the point from which the data series zooms in and out. For example, set the `horizontalFocus` property to `left` and the `verticalFocus` property to `top` to have the series data zoom to and from the upper-left corner of either element or the chart (depending on the setting of the `relativeTo` property).<br>Valid values of the `horizontalFocus` property are `left`, `center`, `right`, and `undefined`.<br>Valid values of the `verticalFocus` property are `top`, `center`, `bottom`, and `undefined`.<br>If you specify only one of these two properties, the focus is a horizontal or vertical line rather than a point. For example, when you set the `horizontalFocus` property to `left`, the element zooms to and from a vertical line along the left edge of its bounding box. Setting the `verticalFocus` property to `center` causes the element to zoom to and from a horizontal line along the middle of its bounding box.<br>The default value for both properties is `center`. |
| `relativeTo` | Controls the bounding box used to calculate the focal point of the zooms. Valid values for `relativeTo` are `series` and `chart`.<br>Set the `relativeTo` property to `series` to zoom each element relative to itself. For example, each column of a ColumnChart zooms from the upper-left of the column.<br>Set the `relativeTo` property to `chart` to zoom each element relative to the chart area. For example, each column zooms from the upper-left of the axes, the center of the axes, and so on. |

The following example zooms in the data series from the upper-right corner of the chart. While zooming in, Flex displays the last element in the series first because the `elementOffset` value is negative.

```
<?xml version="1.0"?>
<!-- charts/SeriesZoomEffect.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var items:ArrayCollection = new ArrayCollection([
        {item: 2000},
        {item: 3300},
        {item: 3000},
        {item: 2100},
        {item: 3200}
     ]);

     public function addDataItem():void {
        // Add a randomly generated value to the data provider
        var n:Number = Math.random() * 3000;
        var o:Object = {item: n};
        items.addItem(o);
     }
  ]]></mx:Script>

  <!-- Define chart effects -->
  <mx:SeriesZoom id="zoomOut"
     duration="2000"
     minimumElementDuration="50"
     elementOffset="50"
     verticalFocus="top"
     horizontalFocus="left"
     relativeTo="chart"
  />
  <mx:SeriesZoom id="zoomIn"
     duration="2000"
     minimumElementDuration="50"
     elementOffset="-50"
     verticalFocus="top"
     horizontalFocus="right"
     relativeTo="chart"
  />

  <mx:Panel title="Column Chart with Series Zoom Effect">
     <mx:ColumnChart id="myChart" dataProvider="{items}">
        <mx:series>
           <mx:ColumnSeries
            yField="item"
            displayName="Quantity"
            showDataEffect="zoomIn"
            hideDataEffect="zoomOut"
           />
        </mx:series>
     </mx:ColumnChart>
  </mx:Panel>
  <mx:Button id="b1" click="addDataItem()" label="Add Data Item"/>
</mx:Application>
```

When you use the SeriesZoom effect, the entire data series disappears from the chart when the effect begins. The data then reappears based on the nature of the effect. To have the data stay on the screen at all times during the effect, you can use the SeriesInterpolate effect. For more information, see "Using the SeriesInterpolate effect" on page 1911.

## Using the SeriesInterpolate effect

The SeriesInterpolate effect moves the graphics that represent the existing data in the series to the new points. Instead of clearing the chart and then repopulating it as with SeriesZoom and SeriesSlide, this effect keeps the data on the screen at all times.

You use the SeriesInterpolate effect only with the `showDataEffect` effect trigger. It has no effect if set with a `hideDataEffect` trigger.

The following example sets the `elementOffset` property of SeriesInterpolate to 0. As a result, all elements move to their new locations without disappearing from the screen.

```
<?xml version="1.0"?>
<!-- charts/SeriesInterpolateEffect.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var items:ArrayCollection = new ArrayCollection([
        {item: 2000},
        {item: 3300},
        {item: 3000},
        {item: 2100},
        {item: 3200}
     ]);

     public function addDataItem():void {
        // Add a randomly generated value to the data provider
        var n:Number = Math.random() * 3000;
        var o:Object = {item: n};
        items.addItem(o);
     }
  ]]></mx:Script>

  <!-- Define chart effect -->
  <mx:SeriesInterpolate id="rearrangeData"
     duration="1000"
     minimumElementDuration="200"
     elementOffset="0"
  />

  <mx:Panel title="Column Chart with Series Interpolate Effect">
     <mx:ColumnChart id="myChart" dataProvider="{items}">
        <mx:series>
           <mx:ColumnSeries
            yField="item"
            displayName="Quantity"
            showDataEffect="rearrangeData"
           />
        </mx:series>
     </mx:ColumnChart>
  </mx:Panel>
  <mx:Button id="b1" click="addDataItem()" label="Add Data Item"/>
</mx:Application>
```

## Applying effects with ActionScript

You can define effects and apply them to chart series by using ActionScript. One way to apply an effect is the same as the way you apply style properties. You use the `setStyle()` method and Cascading Style Sheets (CSS) to apply the effect. For more information on using styles, see Chapter 20, "Using Styles and Themes," on page 767.

The following example defines the `slideIn` effect that plays every time the user adds a data item to the chart control's ColumnSeries. The effect is applied to the series by using the `setStyle()` method when the application first loads.

```
<?xml version="1.0"?>
<!-- charts/ApplyEffectsAsStyles.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="init()">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     import mx.charts.effects.SeriesInterpolate;

     [Bindable]
     public var items:ArrayCollection = new ArrayCollection([
        {item: 2000},
        {item: 3300},
        {item: 3000},
        {item: 2100},
        {item: 3200}
     ]);

     public var rearrangeData:SeriesInterpolate;

     public function init():void {
        rearrangeData = new SeriesInterpolate();
        rearrangeData.duration = 1000;
        rearrangeData.minimumElementDuration = 200;
        rearrangeData.elementOffset = 0;

        // Apply the effect as a style.
        mySeries.setStyle("showDataEffect", "rearrangeData");
     }

     public function addDataItem():void {
        // Add a randomly generated value to the data provider.
        var n:Number = Math.random() * 3000;
        var o:Object = {item: n};
        items.addItem(o);
     }
  ]]></mx:Script>

  <mx:Panel title="Column Chart with Series Interpolate Effect">
     <mx:ColumnChart id="myChart" dataProvider="{items}">
        <mx:series>
           <mx:ColumnSeries
            id="mySeries"
            yField="item"
            displayName="Quantity"
            showDataEffect="rearrangeData"
           />
        </mx:series>
     </mx:ColumnChart>
  </mx:Panel>

  <mx:Button id="b1"
    click="addDataItem()"
    label="Add Data Item"
  />

</mx:Application>
```

When you define an effect in ActionScript, you must ensure that you import the appropriate classes. If you define an effect by using MXML, the compiler imports the class for you.

Instead of applying effects with the `setStyle()` method, you can apply them with CSS if you predefine them in your MXML application; for example:

```
<?xml version="1.0"?>
<!-- charts/ApplyEffectsWithCSS.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

  <mx:Style>
     ColumnSeries {
        showDataEffect:slideDown;
        hideDataEffect:slideDown;
     }
  </mx:Style>

  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [Bindable]
     public var items:ArrayCollection = new ArrayCollection([
        {item:2000},
        {item:3300},
        {item:3000},
        {item:2100},
        {item:3200}
     ]);

     public function addDataItem():void {
        // Add a randomly generated value to the data provider
        var n:Number = Math.random() * 3000;
        var o:Object = {item:n};
        items.addItem(o);
     }
  ]]></mx:Script>

  <!-- Define chart effect -->
  <mx:SeriesSlide
    duration="1000"
    direction="down"
    minimumElementDuration="20"
    elementOffset="30"
    id="slideDown"
  />

  <mx:Panel title="Column Chart with Custom Series Slide Effect">
     <mx:ColumnChart id="myChart" dataProvider="{items}">
        <mx:series>
           <mx:ColumnSeries
            yField="item"
            displayName="Quantity"
           />
        </mx:series>
     </mx:ColumnChart>
  </mx:Panel>
  <mx:Button id="b1"
    click="addDataItem()"
    label="Add Data Item"
  />
</mx:Application>
```

You can also apply effects with ActionScript and not use styles. You do this by specifying the target of the effect (the series) in the effect's constructor, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/ApplyEffectsWithActionScript.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="init()">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     import mx.charts.effects.SeriesInterpolate;

     [Bindable]
     public var items:ArrayCollection = new ArrayCollection([
        {item: 2000},
        {item: 3300},
        {item: 3000},
        {item: 2100},
        {item: 3200}
     ]);

     public var rearrangeData:SeriesInterpolate;

     public function init():void {
        // Specify the effect's target in the constructor.
        rearrangeData = new SeriesInterpolate(mySeries);
        rearrangeData.duration = 1000;
        rearrangeData.minimumElementDuration = 200;
        rearrangeData.elementOffset = 0;
     }

     public function addDataItem():void {
        // Add a randomly generated value to the data provider.
        var n:Number = Math.random() * 3000;
        var o:Object = {item: n};
        items.addItem(o);
     }
  ]]></mx:Script>

  <mx:Panel title="Column Chart with Series Interpolate Effect">
     <mx:ColumnChart id="myChart" dataProvider="{items}">
        <mx:series>
           <mx:ColumnSeries
            id="mySeries"
            yField="item"
            displayName="Quantity"
            showDataEffect="rearrangeData"
           />
        </mx:series>
     </mx:ColumnChart>
  </mx:Panel>
  <mx:Button id="b1"
    click="addDataItem()"
    label="Add Data Item"
  />

</mx:Application>
```

You can expand on this example to use slider controls to adjust the effect's properties. To bind the properties of an ActionScript object, such as an effect, to a control, you use methods of the BindingUtils class, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/ApplyEffectsWithActionScriptSlider.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="init()">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     import mx.charts.effects.SeriesInterpolate;
     import mx.binding.utils.BindingUtils;

     public var rearrangeData:SeriesInterpolate;
     [Bindable]
     public var items:ArrayCollection = new ArrayCollection([
        {item: 2000},
        {item: 3300},
        {item: 3000},
        {item: 2100},
        {item: 3200}
     ]);

     public function init():void {
        // Specify the effect's target in the constructor.
        rearrangeData = new SeriesInterpolate(mySeries);

        // Bind effect properties to slider controls
        BindingUtils.bindProperty(rearrangeData,
            "duration", durationSlider, "value");
        BindingUtils.bindProperty(rearrangeData,
            "minimumElementDuration",
            minimumElementDurationSlider, "value");
        BindingUtils.bindProperty(rearrangeData,
            "elementOffset", elementOffsetSlider, "value");
     }

     public function addDataItem():void {
        // Add a randomly generated value to the data provider.
        var n:Number = Math.random() * 3000;
        var o:Object = {item: n};
        items.addItem(o);
     }

     public function resetSliders():void {
        durationSlider.value = 1000;
        minimumElementDurationSlider.value = 200;
        elementOffsetSlider.value = 0;
     }
  ]]></mx:Script>
```

```
<mx:Panel title="Column Chart with Series Interpolate Effect">
   <mx:ColumnChart id="myChart" dataProvider="{items}">
      <mx:series>
         <mx:ColumnSeries
          id="mySeries"
          yField="item"
          displayName="Quantity"
          showDataEffect="rearrangeData"
         />
      </mx:series>
   </mx:ColumnChart>
</mx:Panel>

<mx:Button id="b1"
  click="addDataItem()"
  label="Add Data Item"
/>

<mx:Panel>
   <mx:Form>
      <mx:FormItem label="Duration">
         <mx:HSlider id="durationSlider"
          minimum="1"
          maximum="10000"
          value="1000"
          dataTipPlacement="top"
          tickColor="black"
          snapInterval="500"
          tickInterval="500"
          labels="['0','10000']"
          allowTrackClick="true"
          liveDragging="true"
         />
      </mx:FormItem>
      <mx:FormItem label="Minimum Element Duration">
         <mx:HSlider id="minimumElementDurationSlider"
          minimum="0"
          maximum="1000"
          value="200"
          dataTipPlacement="top"
          tickColor="black"
          snapInterval="50"
          tickInterval="50"
          labels="['0','1000']"
          allowTrackClick="true"
          liveDragging="true"
         />
      </mx:FormItem>
      <mx:FormItem label="Element Offset">
```

```
            <mx:HSlider id="elementOffsetSlider"
             minimum="0"
             maximum="1000"
             value="0"
             dataTipPlacement="top"
             tickColor="black"
             snapInterval="50"
             tickInterval="50"
             labels="['0','1000']"
             allowTrackClick="true"
             liveDragging="true"
            />
        </mx:FormItem>
     </mx:Form>
  </mx:Panel>

  <mx:Button id="b2"
    label="Reset Sliders"
    click="resetSliders()"
  />

</mx:Application>
```

For more information about databinding in ActionScript, see “Defining data bindings in ActionScript” on page 1506.

## Drilling down into data

One common use of charts is to allow the user to drill down into the data. This usually occurs when the user performs some sort of event on the chart such as clicking on a wedge in a PieChart control or clicking on a column in a ColumnChart control. Clicking on a data item reveals a new chart that describes the make-up of that data item.

For example, you might have a ColumnChart control that shows the month-by-month production of widgets. To initially populate this chart, you might make a database call (through a service or some other adapter). If the user then clicks on the January column, the application could display the number of widgets of each color that were produced that month. To get the individual month's widget data, you typically make another database call and pass a parameter to the listening service that describes the specific data you want. You can then use the resulting data provider to render the new view.

Typically, when you drill down into chart data, you create new charts in your application with ActionScript. When creating new charts in ActionScript, you must be sure to create a series, add it to the new chart's series Array, and then call the `addChild()` method on the container to add the new chart to the display list. For more information, see "Creating charts in ActionScript" on page 1605.

One way to provide drill-down functionality is to make calls that are external to the application to get the drill-down data. You typically do this by using the chart's `itemClick` event listener, which gives you access to the HitData object. The HitData object lets you examine what data was underneath the mouse when the event was triggered. This capability lets you perform actions on specific chart data. For more information, see "Using the HitData object" on page 1883.

You can also use a simple Event object to get a reference to the series that was clicked. The following example shows the net worth of a fictional person. When you click on a column in the initial view, the example drills down into a second view that shows the change in value of a particular asset class over time.

The following example uses the Event object to get a reference to the clicked ColumnSeries. It then drills down into the single ColumnSeries by replacing the chart's series Array with the single ColumnSeries in the chart. When you click a column again, the chart returns to its original configuration with all ColumnSeries.

```
<?xml version="1.0"?>
<!-- charts/SimpleDrillDown.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" height="100%"
width="100%" creationComplete="initApp()">
    <mx:Script><![CDATA[
        import mx.collections.ArrayCollection;

        [Bindable]
        public var dpac:ArrayCollection = new ArrayCollection ([
            { date:"01/01/2006", cash:50000,
              stocks:198192, retirement:130101,
              home:750000, other:19148 },
            { date:"02/01/2006", cash:50000,
              stocks:210309, retirement:143707,
              home:760000, other:19493 },
            { date:"03/01/2006", cash:50000,
              stocks:238992, retirement:169529,
              home:770000, other:19933 },
            { date:"04/01/2006", cash:50000,
              stocks:292269, retirement:242596,
              home:770000, other:21445 }]);

        public var initSeriesArray:Array = new Array();
        public var level:Number = 1;
        public var newSeries:Array;

        private function initApp():void {
            // Get initial series Array -- to be reloaded when it returns
            // from a drill down.
            initSeriesArray = chart.series;
        }

        private function zoomIntoSeries(e:Event):void {
            newSeries = new Array();
            if (level == 1) {
                newSeries.push(e.currentTarget);
                level = 2;
            } else {
                newSeries = initSeriesArray;
                p1.title = "Net Worth";
                level = 1;
            }
            chart.series = newSeries;
        }
    ]]></mx:Script>

    <mx:Panel id="p1" title="Net Worth">
        <mx:ColumnChart id="chart"
            dataProvider="{dpac}"
            type="stacked"
            showDataTips="true"
        >
            <mx:series>
                <mx:ColumnSeries id="s1"
                    displayName="Cash"
                    yField="cash"
                    xField="date"
                    click="zoomIntoSeries(event)"
                />
                <mx:ColumnSeries id="s2"
                    displayName="Stocks"
                    yField="stocks"
                    xField="date"
                    click="zoomIntoSeries(event)"
                />
                <mx:ColumnSeries id="s3"
                    displayName="Retirement"
                    yField="retirement"
                    xField="date"
                    click="zoomIntoSeries(event)"
                />
                <mx:ColumnSeries id="s4"
                    displayName="Home"
                    yField="home"
                    xField="date"
                    click="zoomIntoSeries(event)"
                />
                <mx:ColumnSeries id="s5"
                    displayName="Other"
                    yField="other"
                    xField="date"
                    click="zoomIntoSeries(event)"
                />
            </mx:series>
            <mx:horizontalAxis >
                <mx:DateTimeAxis title="Date" dataUnits="months"/>
            </mx:horizontalAxis>
        </mx:ColumnChart>
        <mx:Legend dataProvider="{chart}"/>
    </mx:Panel>
</mx:Application>
```

Another approach to drilling down into chart data is to use unused data within the existing data provider. You can do this by changing the properties of the series and axes when the chart is clicked.

The following example is similar to the previous example in that it drills down into the assets of a fictional person's net worth. In this case, though, it shows the value of the asset classes for the clicked-on month in the drill-down view rather than the change over time of a particular asset class.

This example uses the HitData object's `item` property to access the values of the current data provider. By building an Array of objects with the newly-discovered data, the chart is able to drill down into the series without making calls to any external services.

Drilling down into data is an ideal time to use effects such as SeriesSlide. This example also defines `seriesIn` and `seriesOut` effects to slide the columns in and out when the drilling down (and the return from drilling down) occurs.

```
<?xml version="1.0"?>
<!-- charts/DrillDownWithEffects.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" height="100%"
width="100%">
    <mx:Script><![CDATA[
        import mx.collections.ArrayCollection;
        import mx.charts.HitData;
        import mx.charts.events.ChartItemEvent;

        [Bindable]
        public var dpac:ArrayCollection = new ArrayCollection ([
           { date:"01/01/2006", assets:1607441, cash:520000, stocks:98192,
               retirement:130101, home:850000, other:9148 },
           { date:"02/01/2006", assets:1610509, cash:520000, stocks:97309,
               retirement:133707, home:850000, other:9493 },
           { date:"03/01/2006", assets:1617454, cash:520000, stocks:97992,
               retirement:139529, home:850000, other:9933 },
           { date:"04/01/2006", assets:1615310, cash:520000, stocks:92269,
               retirement:142596, home:850000, other:10445 },
           { date:"05/01/2006", assets:1600304, cash:520000, stocks:80754,
               retirement:139029, home:850000, other:10521 },
           { date:"06/01/2006", assets:1600416, cash:520000, stocks:80667,
               retirement:139024, home:850000, other:10725 },
           { date:"07/01/2006", assets:1599340, cash:520000, stocks:78913,
               retirement:139265, home:850000, other:11162 },
           { date:"08/01/2006", assets:1608965, cash:520000, stocks:84754,
               retirement:142618, home:850000, other:11593 },
           { date:"09/01/2006", assets:1622719, cash:520000, stocks:91078,
               retirement:149257, home:850000, other:12384 },
           { date:"10/01/2006", assets:1629806, cash:520000, stocks:86452,
               retirement:160310, home:850000, other:13044 },
           { date:"11/01/2006", assets:1642285, cash:520000, stocks:92172,
               retirement:166357, home:850000, other:13756 },
           { date:"12/01/2006", assets:1651009, cash:520000, stocks:95095,
               retirement:171557, home:850000, other:14357 }]);

        [Bindable]
        public var drillDownDataSet:ArrayCollection;

        [Bindable]
        public var dp:ArrayCollection = dpac;

        private function zoomIntoSeries(e:ChartItemEvent):void {
            if (dp==dpac) {
                drillDownDataSet = new ArrayCollection(genData(e));
                cs1.displayName = "Assets";
               cs1.yField = "amount";
               cs1.xField = "type";

               ca1.categoryField = "type";

               p1.title = "Asset breakdown for " + e.hitData.item.date;
               dp = drillDownDataSet;

              // Remove the Legend. It is not needed in this view because
               // each asset class is its own column in the drill down.
               p1.removeChild(myLegend);
            } else {
               cs1.displayName = "All Assets";
               cs1.yField = "assets";
               cs1.xField = "date";

               ca1.categoryField = "date";

               p1.title = "Net Worth";
               dp = dpac;

               // Add the Legend back to the Panel.
               p1.addChild(myLegend);
            }
        }

        private function genData(e:ChartItemEvent):Array {
               var result:Array = [];

               trace("Cash: " + e.hitData.item.cash);
               trace("Stocks: " + e.hitData.item.stocks);
               trace("Retirement: " + e.hitData.item.retirement);
               trace("Home: " + e.hitData.item.home);
               trace("Other: " + e.hitData.item.other);

               var o1:Object = {
                   type:"cash",
                   amount:e.hitData.item.cash
               };
               var o2:Object = {
                   type:"stocks",
                   amount:e.hitData.item.stocks
               };
               var o3:Object = {
                   type:"retirement",
                   amount:e.hitData.item.retirement
               };
               var o4:Object = {
                   type:"home",
                   amount:e.hitData.item.home
               };
               var o5:Object = {
                   type:"other",
                   amount:e.hitData.item.other
               };
               trace(o1.type + ":" + o1.amount);

               result.push(o1);
               result.push(o2);
               result.push(o3);
               result.push(o4);
               result.push(o5);
               return result;
        }

    ]]></mx:Script>

    <mx:SeriesSlide id="slideIn" duration="1000" direction="down"/>
    <mx:SeriesSlide id="slideOut" duration="1000" direction="up"/>

    <mx:Panel id="p1" title="Net Worth">
        <mx:ColumnChart id="chart"
            showDataTips="true"
            itemClick="zoomIntoSeries(event)"
            dataProvider="{dp}"
        >
            <mx:series>
               <mx:ColumnSeries id="cs1"
                   displayName="All Assets"
                   yField="assets"
                   xField="date"
                   hideDataEffect="slideOut"
                   showDataEffect="slideIn"/>
            </mx:series>

            <mx:horizontalAxis>
               <mx:CategoryAxis id="ca1" categoryField="date"/>
            </mx:horizontalAxis>
        </mx:ColumnChart>

        <mx:Legend id="myLegend" dataProvider="{chart}"/>
    </mx:Panel>
</mx:Application>
```

Another way to drill down into chart data is to use the selection API. You can select a subset of data points on a chart to create a new data provider. For more information, see “Selecting chart items” on page 1930.

## Selecting chart items

As part of interacting with charts, you can select data points (ChartItem objects), examine the underlying data, and then perform actions on those objects. A ChartItem class represents a single entry in the chart series's data provider. A series is made up of an Array of ChartItem objects.

To enable data point selection, you set the value of the `selectionMode` property on the chart. Possible values of this property are `none`, `single`, and `multiple`. Setting the `selectionMode` property to `none` prevents any data points in the chart from being selected. Setting `selectionMode` to `single` lets you select one data point at a time. Setting `selectionMode` to `multiple` lets you select one or more data points at a time. The default value is `none`.

You also toggle the series's selectability by setting the value of the `selectable` property. As a result, while you might set the `selectionMode` on the chart to `multiple`, you can make the data points in some series selectable and others not selectable within the same chart by using the series' `selectable` property.

### Methods of chart item selection

You can programmatically select data points or the application user can interactively select data points in the following ways:

- Mouse selection — Move the mouse pointer over a data point and click the left mouse button to select data points. For more information, see "Keyboard and mouse selection" on page 1934.
- Keyboard selection — You can use the keyboard to select one or more data points, as described in "Keyboard and mouse selection" on page 1934.
- Region selection — Draw a rectangle on the chart. This rectangle defines the range, and selects all data points within that range. For more information, see "Region selection" on page 1933.
- Programmatic selection — Programmatically select one or more data points using the chart selection API. For more information, see "Programmatic selection" on page 1935.

When understanding chart item selection, you should first understand the following terms:

*caret* — The current chart item that could be selected or deselected if you press the space bar. If you select multiple items, one at a time, the caret is the last item selected. If you selected multiple items by using a range, then the caret is the last item in the last series in the selection. This is not always the item that was selected last.

*anchor* — The starting chart item for a multiple selection operation.

The following example creates three different types of charts. You can select one or more data points in each chart using the mouse, keyboard and region selection techniques. The example displays the data points in each series that are currently selected:

```
<?xml version="1.0" ?>
<!--  charts/SimpleSelection.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
    <![CDATA[
        import mx.collections.ArrayCollection;

        private function handleChange(event:Event):void {
            var allSeries:Array = event.currentTarget.series;
            textArea1.text = "";
            for (var i:int=0; i<allSeries.length; i++) {
                textArea1.text += "\n" + allSeries[i].id +
                    " Selected Items: " + allSeries[i].selectedIndices;
            }
        }

        [Bindable]
        public var expenses:ArrayCollection = new ArrayCollection([
            { Expense:"Taxes", Amount:2000 },
            { Expense:"Rent", Amount:1000 },
            { Expense:"Food", Amount:200 } ]);

        [Bindable]
        private var medalsAC:ArrayCollection = new ArrayCollection( [
            { Country: "A", Gold: 35, Silver:39, Bronze: 29 },
            { Country: "B", Gold: 32, Silver:17, Bronze: 14 },
            { Country: "C", Gold: 27, Silver:27, Bronze: 38 },
            { Country: "D", Gold: 15, Silver:15, Bronze: 10 },
            { Country: "E", Gold: 15, Silver:10, Bronze: 10 } ]);

        [Bindable]
        private var profitsAC:ArrayCollection = new ArrayCollection( [
            { Month: "Jan", Profit: 2000, Expenses: 1500, Amount: 450 },
            { Month: "Feb", Profit: 1000, Expenses: 200, Amount: 600 },
            { Month: "Mar", Profit: 1500, Expenses: 500, Amount: 300 },
            { Month: "Apr", Profit: 1800, Expenses: 1200, Amount: 900 },
            { Month: "May", Profit: 2400, Expenses: 575, Amount: 500 } ]);

    ]]>
    </mx:Script>
    <mx:HBox>
        <mx:Panel title="Bar Chart">
            <mx:BarChart id="myBarChart"
                height="225"
                showDataTips="true"
                dataProvider="{medalsAC}"
                selectionMode="multiple"
                change="handleChange(event)"
            >
                <mx:verticalAxis>
                    <mx:CategoryAxis categoryField="Country"/>
                </mx:verticalAxis>
                <mx:series>
                    <mx:BarSeries id="barSeries1"
                        yField="Country"
                        xField="Gold"
                        displayName="Gold"
                        selectable="true"
                    />
                    <mx:BarSeries id="barSeries2"
                        yField="Country"
                        xField="Silver"
                        displayName="Silver"
                        selectable="true"
                    />
                    <mx:BarSeries id="barSeries3"
                        yField="Country"
                        xField="Bronze"
                        displayName="Bronze"
                        selectable="true"
                    />
                </mx:series>
            </mx:BarChart>
            <mx:Legend dataProvider="{myBarChart}"/>
        </mx:Panel>

        <mx:Panel title="Pie Chart" >
            <mx:PieChart id="myPieChart"
                height="225"
                dataProvider="{expenses}"
                showDataTips="true"
                selectionMode="multiple"
                change="handleChange(event)"
            >
                <mx:series>
                    <mx:PieSeries
                        id="pieSeries1"
                        field="Amount"
                        nameField="Expense"
                        labelPosition="callout"
                    />
                </mx:series>
            </mx:PieChart>
            <mx:Legend dataProvider="{myPieChart}"/>
        </mx:Panel>
        <mx:Panel title="Plot Chart">
```

```
            <mx:PlotChart id="myPlotChart"
                height="225"
                showDataTips="true"
                dataProvider="{profitsAC}"
                selectionMode="multiple"
                change="handleChange(event)"
            >
                <mx:series>
                    <mx:PlotSeries id="plotSeries1"
                        xField="Expenses"
                        yField="Profit"
                        displayName="Expenses/Profit"
                        selectable="true"
                    />
                    <mx:PlotSeries id="plotSeries2"
                        xField="Amount"
                        yField="Expenses"
                        displayName="Amount/Expenses"
                        selectable="true"
                    />
                    <mx:PlotSeries id="plotSeries3"
                        xField="Profit"
                        yField="Amount"
                        displayName="Profit/Amount"
                        selectable="true"
                    />
                </mx:series>
            </mx:PlotChart>
            <mx:Legend dataProvider="{myPlotChart}"/>
        </mx:Panel>
    </mx:HBox>
    <mx:HBox width="370" height="71">
        <mx:Label text="Selection Changed Event:" />
        <mx:TextArea id="textArea1" height="70" width="213"/>
    </mx:HBox>
</mx:Application>
```

## Region selection

Users can select all data points in an area on a chart by drawing a rectangle (the *region*) on the chart. Users define a rectangular region by clicking and holding the mouse button while they move the mouse, drawing a rectangle on the chart. On the MOUSE_UP event, the items inside the rectangular region are selected. The starting point for a rectangular range must be over the chart. You cannot start the rectangle outside of the chart control's bounds.

To select data points with region selection, the value of the chart's `selectionMode` property must be `multiple`. If it is `single` or `none`, then you cannot draw the selection rectangle on the chart.

For most charts, as long as any point inside the rectangle touches a data point (for example, a column in a ColumnChart control), you select that data point. For BubbleChart and PieChart controls, an item is selected only if its center is inside the selection rectangle.

## Keyboard and mouse selection

To select chart items, you can use the arrow keys, space bar, and enter key on your keyboard or the mouse pointer. If you want to select more than data point, the value of the chart's `selectionMode` property must be `multiple`.

### Left and right arrow keys

You use the left and right arrows to move up and down the data in the series.

Click the left arrow to select the previous item and the right arrow to select the next item in the series. When you reach the last item in a series, you move to the first item in the next series.

### Up and down arrow keys

Click the up and down arrow keys to move to the next or previous series on the chart.

Click the up arrow key to select the next series in the chart's series array. Click the down arrow key to select the previous series.

The index of the item in each series remains the same. If there is only one series in the chart, then click the up arrow key to select the first data point in the series; click the down arrow key to select the last data point in the series.

### Shift and control keys

You can use the shift and control keys in conjunction with the other keys to select or deselect data points and change the caret in the selection.

Hold the shift key down while clicking the right arrow key to add the next data point to the selection until you reach the end of the series. Click the left arrow key to remove the last data point from the selection.

Hold the shift key down while clicking the up arrow key to select the first item in the next series. Click the down arrow key to select the last item in the previous series.

Hold the control key down while using the arrow keys to move the caret while not deselecting the anchor item. You can then select the new caret by pressing the space bar.

### Space bar

Click the space bar to toggle the selection of the caret item, if the control key is depressed. A deselected caret item appears highlighted, but the highlight color is not the same as the selected item color.

### Page Up/Home and Page Down/End keys

Click the Page Up/Home and Page Down/End keys to move to the first and last items in the current series, respectively. Moving to an item makes that item the caret item, but does not select it. You can press the space bar to select it.

### Mouse pointer

The default behavior of the mouse pointer is to select the data point under the mouse pointer when you click the mouse button and de-select all other data points.

If you click the mouse button and drag it over a chart, you create a rectangular region that defines a selection range. All data points inside that range are selected.

If you select a data point, then hold the shift key down and click on another data point, you select the first point, the target point, and all points that appear inside the rectangular range that you just created. If you select a data point, then hold the control key down and click another data point, you select both data points, but not the data points in between. You can add more individual data points by continuing to hold the control key down while you select another data point.

If you click anywhere on the chart that does not have a data point without any keys held down, then you clear the selection. Clicking outside of the chart control's boundary does not clear the selection.

## Programmatic selection

You can use the chart selection APIs to programmatically select one or more data points in a chart control. These APIs consist of methods and properties of the ChartBase, ChartItem, and chart series objects.

Selections with multiple items include a caret and an anchor. You can access these items by using the `caretItem` and `anchorItem` properties of the chart control.

### Properties of the series

The series defines which ChartItem objects are selected. You can programmatically select items by setting the values of the following properties of the series:

- `selectedItem`

- `selectedItems`
- `selectedIndex`
- `selectedIndices`

The index properties refer to the index of the chart item in the series. This index is the same as the index in the data provider, assuming you did not sort or limit the series items.

Programmatically setting the values of these properties does not trigger a `change` event.

The following example increments and decrements the series's `selectedIndex` property to select each item, one after the other, in the series:

```
<?xml version="1.0" ?>
<!--  charts/SimplerCycle.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
    <mx:Script>
    <![CDATA[
        import mx.collections.ArrayCollection;
        import mx.charts.chartClasses.ChartBase;
        import mx.charts.ChartItem;
        import mx.charts.series.items.ColumnSeriesItem;

        [Bindable]
        private var expensesAC:ArrayCollection = new ArrayCollection( [
            { Month: "Jan", Expenses: 1500, Amount: 450, Profit: 2000 },
            { Month: "Feb", Expenses: 200, Amount: 600, Profit: 1000 },
            { Month: "Mar", Expenses: 500, Amount: 300, Profit: 1500 },
            { Month: "Apr", Expenses: 1200, Amount: 900, Profit: 1800 },
            { Month: "May", Expenses: 575, Amount: 500, Profit: 2400 } ]);

        private function initApp():void {
            // Select the first item on start up.
            series1.selectedIndex = 0;
        }

        private function getNext(e:Event):void {
            series1.selectedIndex += 1;
        }

        private function getPrev(e:Event):void {
            series1.selectedIndex -= 0;
        }

        private function getFirst(e:Event):void {
            series1.selectedIndex = 0;
        }

        private function getLast(e:Event):void {
            series1.selectedIndex = series1.items.length - 1;
        }

    ]]>
    </mx:Script>
    <mx:Panel height="100%" width="100%">
        <mx:ColumnChart id="myChart" height="207" width="350"
showDataTips="true" dataProvider="{expensesAC}" selectionMode="single">
            <mx:series>
                <mx:ColumnSeries id="series1" yField="Expenses"
```

```
displayName="Expenses" selectable="true"/>
            </mx:series>

            <mx:horizontalAxis>
                <mx:CategoryAxis categoryField="Month"/>
            </mx:horizontalAxis>

        </mx:ColumnChart>

        <mx:HBox>
            <mx:Label id="label0" text="Value: "/>
            <mx:Label id="label1"/>
        </mx:HBox>

        <mx:Legend dataProvider="{myChart}" width="200"/>

        <mx:HBox>
            <mx:Button label="|&lt;" click="getFirst(event);" />
            <mx:Button label="&lt;" click="getPrev(event);" />
            <mx:Button label="&gt;" click="getNext(event);" />
            <mx:Button label="&gt;|" click="getLast(event);" />
        </mx:HBox>
    </mx:Panel>
</mx:Application>
```

To cycle through ChartItem objects in a series, you can use methods such as `getNextItem()` and `getPreviousItem()`. For more information, see “Methods and properties of the ChartBase class” on page 1942.

The `selectedIndices` property lets you select any number of ChartItems in a chart control. The following example uses the `selectedIndices` property to select all items in all series when the user presses the Ctrl+a keys on the keyboard:

```
<?xml version="1.0" ?>
<!--  charts/SelectAllItems.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
    <mx:Script>
    <![CDATA[
        import mx.collections.ArrayCollection;
        import flash.events.KeyboardEvent;
        import mx.charts.events.ChartItemEvent;

        [Bindable]
        private var expensesAC:ArrayCollection = new ArrayCollection( [
            { Month: "Jan", Profit: 2000, Expenses: 1500, Amount: 450 },
            { Month: "Feb", Profit: 1000, Expenses: 200, Amount: 600 },
            { Month: "Mar", Profit: 1500, Expenses: 500, Amount: 300 },
            { Month: "Apr", Profit: 1800, Expenses: 1200, Amount: 900 },
            { Month: "May", Profit: 2400, Expenses: 575, Amount: 500 } ]);

        private function initApp():void {
            application.addEventListener(KeyboardEvent.KEY_UP, keyHandler);
        }

        private function keyHandler(event:KeyboardEvent):void {
            var ctrlPressed:Boolean = event.ctrlKey;

            // If the user presses Ctrl + A, select all chart items.
            if (ctrlPressed) {
                var curKeyCode:int = event.keyCode;
                if (curKeyCode == 65) { // 65 is the keycode value for 'a'
                    selectItems();
                }
            }
        }

        private function selectItems():void {
            // Create an array of all the chart's series.
            var allSeries:Array = myChart.series;

            // Iterate over each series.
            for (var i:int=0; i<allSeries.length; i++) {
                var selectedData:Array = [];

                // Iterate over the number of items in the series.
                for (var j:int=0; j<expensesAC.length; j++) {
                        selectedData.push(j);
                }
               // Use the series' selectedIndices property to select all the
                // chart items.
                allSeries[i].selectedIndices = selectedData;
            }
        }
    ]]>
    </mx:Script>
    <mx:Panel height="100%" width="100%">
        <mx:PlotChart id="myChart"
            height="207"
            width="350"
            showDataTips="true"
            dataProvider="{expensesAC}"
            selectionMode="multiple"
        >
            <mx:series>
                <mx:PlotSeries id="series1"
                    xField="Expenses"
                    yField="Profit"
                    displayName="Expenses/Profit"
                    selectable="true"
                />
                <mx:PlotSeries id="series2"
                    xField="Amount"
                    yField="Expenses"
                    displayName="Amount/Expenses"
                    selectable="true"
                />
                <mx:PlotSeries id="series3"
                    xField="Profit"
                    yField="Amount"
                    displayName="Profit/Amount"
                    selectable="true"
                />
            </mx:series>
        </mx:PlotChart>
        <mx:Legend dataProvider="{myChart}" width="200"/>

    </mx:Panel>
</mx:Application>
```

The following example is similar to the previous example, except that it lets you set a threshold value in the TextInput control. The chart only selects chart items whose values are greater than that threshold.

```
<?xml version="1.0" ?>
<!--  charts/ConditionallySelectChartItems.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
    <![CDATA[
        import mx.collections.ArrayCollection;

        [Bindable]
        private var expensesAC:ArrayCollection = new ArrayCollection( [
            { Month: "Jan", Profit: 2000, Expenses: 1500, Amount: 450 },
            { Month: "Feb", Profit: 1000, Expenses: 200, Amount: 600 },
            { Month: "Mar", Profit: 1500, Expenses: 500, Amount: 300 },
            { Month: "Apr", Profit: 1800, Expenses: 1200, Amount: 900 },
            { Month: "May", Profit: 2400, Expenses: 575, Amount: 500 } ]);

        private function selectItems(event:Event):void {
            // Create an array of all the chart's series.
            var allSeries:Array = myChart.series;

            // Iterate over each series.
            for (var i:int=0; i<allSeries.length; i++) {
                var selectedData:Array = [];
                // Iterate over each item in the series.
                for (var j:int=0; j<expensesAC.length; j++) {
                    if (expensesAC.getItemAt(j).Profit >=
                        Number(threshold.text)) {
                        selectedData.push(j);
                    }
                }

               // Use the series' selectedIndices property to select all the
                // chart items that met the criteria.
                allSeries[i].selectedIndices = selectedData;
            }
        }
    ]]>
    </mx:Script>
    <mx:Panel height="100%" width="100%">
        <mx:PlotChart id="myChart"
            height="207"
            width="350"
            showDataTips="true"
            dataProvider="{expensesAC}"
            selectionMode="multiple"
        >
            <mx:series>
```

```
                <mx:PlotSeries id="series1"
                    xField="Expenses"
                    yField="Profit"
                    displayName="Expenses/Profit"
                    selectable="true"
                />
                <mx:PlotSeries id="series2"
                    xField="Amount"
                    yField="Expenses"
                    displayName="Amount/Expenses"
                    selectable="true"
                />
                <mx:PlotSeries id="series3"
                    xField="Profit"
                    yField="Amount"
                    displayName="Profit/Amount"
                    selectable="true"
                />
            </mx:series>
        </mx:PlotChart>
        <mx:Legend dataProvider="{myChart}" width="200"/>

        <mx:TextInput id="threshold" text="1000"/>
        <mx:Button label="Select Items" click="selectItems(event)"/>

    </mx:Panel>
</mx:Application>
```

## Methods and properties of the ChartBase class

The ChartBase class is the parent class of all chart controls. You can programmatically access ChartItem objects by using the following methods of this class:

- `getNextItem()`
- `getPreviousItem()`
- `getFirstItem()`
- `getLastItem()`

These methods return a ChartItem object, whether it is selected or not. Which object returned depends on which one is currently selected, and which direction constant (`ChartBase.HORIZONTAL` or `ChartBase.VERTICAL`) you pass to the method.

The following example uses these methods to cycle through the data points in a ColumnChart control. It sets the value of the series's `selectedItem` property to identify the new ChartItem as the currently selected item.

```
<?xml version="1.0" ?>
<!--  charts/SimpleCycle.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
    <mx:Script>
    <![CDATA[
        import mx.collections.ArrayCollection;
        import mx.charts.chartClasses.ChartBase;
        import mx.charts.ChartItem;
        import mx.charts.series.items.ColumnSeriesItem;

        [Bindable]
        private var expensesAC:ArrayCollection = new ArrayCollection( [
            { Month: "Jan", Expenses: 1500, Amount: 450, Profit: 2000 },
            { Month: "Feb", Expenses: 200, Amount: 600, Profit: 1000 },
            { Month: "Mar", Expenses: 500, Amount: 300, Profit: 1500 },
            { Month: "Apr", Expenses: 1200, Amount: 900, Profit: 1800 },
            { Month: "May", Expenses: 575, Amount: 500, Profit: 2400 } ]);

        private function initApp():void {
            // Select the first item on start up.
            series1.selectedIndex = 0;
        }

        private function getNext(e:Event, dir:*):void {
            var curItem:ChartItem = series1.selectedItem;
            var newItem:ChartItem = myChart.getNextItem(dir);
            applyNewItem(newItem);
        }

        private function getPrev(e:Event, dir:*):void {
            var curItem:ChartItem = series1.selectedItem;
            var newItem:ChartItem = myChart.getPreviousItem(dir);
            applyNewItem(newItem);
        }

        private function getFirst(e:Event, dir:*):void {
            var newItem:ChartItem = myChart.getFirstItem(dir);
            applyNewItem(newItem);
        }

        private function getLast(e:Event, dir:*):void {
            var newItem:ChartItem = myChart.getLastItem(dir);
            applyNewItem(newItem);
        }
        private function applyNewItem(n:ChartItem):void {
            series1.selectedItem = n;
        }
    ]]>
    </mx:Script>
    <mx:Panel height="100%" width="100%">
        <mx:ColumnChart id="myChart"
            height="207"
            width="350"
            showDataTips="true"
            dataProvider="{expensesAC}"
            selectionMode="single"
        >
            <mx:series>
                <mx:ColumnSeries id="series1"
                    yField="Expenses"
                    displayName="Expenses"
                    selectable="true"
                />
            </mx:series>
            <mx:horizontalAxis>
                <mx:CategoryAxis categoryField="Month"/>
            </mx:horizontalAxis>
        </mx:ColumnChart>

        <mx:HBox>
            <mx:Label id="label0" text="Value: "/>
            <mx:Label id="label1"/>
        </mx:HBox>

        <mx:Legend dataProvider="{myChart}" width="200"/>

        <mx:HBox>
            <mx:Button label="|&lt;"
                click="getFirst(event, ChartBase.HORIZONTAL);" />
            <mx:Button label="&lt;"
                click="getPrev(event, ChartBase.HORIZONTAL);" />
            <mx:Button label="&gt;"
                click="getNext(event, ChartBase.HORIZONTAL);" />
            <mx:Button label="&gt;|"
                click="getLast(event, ChartBase.HORIZONTAL);" />
        </mx:HBox>
    </mx:Panel>
</mx:Application>
```

These item getter methods do not have associated setters. You cannot set the selected ChartItem objects in the same manner. Instead, you use the properties of the series, as described in “Properties of the series” on page 1935.

Calling the `getNextItem()` and `getPreviousItem()` methods provides more control over the item selection than simply incrementing and decrementing the series' `selectedIndex` property as shown in "Properties of the series" on page 1935. With these methods, you can choose the direction in which you select the next or previous item, and then decide whether to make the item appear selected.

The item getter methods also account for when you get to the end of the series, for example. If no ChartItems are currently selected, then the `getNextItem()` method gets the first one in the series. If you are at the beginning of the series, the `getPreviousItem()` gets the last item in the series.

The ChartBase class defines two additional properties, `selectedChartItem` and `selectedChartItems`, that return an item or Array of items that are selected. You cannot set the values of these properties to select chart items. These properties are read-only. They are useful if your chart has multiple series and you want to know which items across the series are selected without iterating over all the series.

### ChartItem properties

You can set the value of the `currentState` property of a ChartItem object to make it appear selected or deselected, or make it appear in some other state. The `currentState` property can be set to `none`, `rollOver`, `selected`, `disabled`, `focusSelected`, and `focused`.

Setting the state of the item does not add it to the `selectedItems` array. It only changes the appearance of the chart item. Setting the value of this property also does not trigger a `change` event.

### Defined range

You can use the `getItemsInRange()` method to define a rectangular range and select those chart items that are within that range. The `getItemsInRange()` method takes an instance of the Rectangle class as its only argument. This rectangle defines a region on the stage, in global coordinates.

Getting an array of ChartItem objects with the `getItemsInRange()` method does not trigger a `change` event. The state of the items does not change.

The following example lets you specify the `x`, `y`, `height`, and `width` properties of a rectangular range. It then calls the `getItemsInRange()` method and passes those values to define the range. All chart items that fall within this invisible rectangle are selected and their values are displayed in the TextArea control.

```
<?xml version="1.0" ?>
<!--  charts/GetItemsInRangeExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
    <![CDATA[
        import mx.collections.ArrayCollection;
        import mx.charts.chartClasses.ChartBase;
        import mx.charts.ChartItem;
        import mx.charts.series.items.ColumnSeriesItem;

        [Bindable]
        private var expensesAC:ArrayCollection = new ArrayCollection( [
            { Month: "Jan", Expenses: 1500, Amount: 450, Profit: 2000 },
            { Month: "Feb", Expenses: 200, Amount: 600, Profit: 1000 },
            { Month: "Mar", Expenses: 500, Amount: 300, Profit: 1500 },
            { Month: "Apr", Expenses: 1200, Amount: 900, Profit: 1800 },
            { Month: "May", Expenses: 575, Amount: 500, Profit: 2400 } ]);

        private function getItems(e:Event):void {
            var x:Number = Number(ti1.text);
            var y:Number = Number(ti2.text);
            var h:Number = Number(ti3.text);
            var w:Number = Number(ti4.text);

            var r:Rectangle = new Rectangle(x, y, h, w);

            // Get an array of ChartItems in the defined range.
            var a:Array = myChart.getItemsInRange(r);

            for (var i:int = 0; i<a.length; i++) {
                var myChartItem:ColumnSeriesItem = ColumnSeriesItem(a[i]);

                // Make items appear selected.
                myChartItem.currentState = "selected";

                // Show values of the items that appear selected.
                ta1.text += myChartItem.xValue.toString() +
                    "=" + myChartItem.yValue.toString() + "\n";
            }
        }
    ]]>
    </mx:Script>
    <mx:Panel id="p1" height="100%" width="100%">
        <mx:ColumnChart id="myChart"
            height="207"
            width="350"
            showDataTips="true"
            dataProvider="{expensesAC}"
            selectionMode="multiple"
        >
            <mx:series>
                <mx:ColumnSeries id="series1"
                    yField="Expenses"
                    displayName="Expenses"
                    selectable="true"
                />
                <mx:ColumnSeries id="series2"
                    yField="Amount"
                    displayName="Amount"
                    selectable="true"
                />
                <mx:ColumnSeries id="series3"
                    yField="Profit"
                    displayName="Profit"
                    selectable="true"
                />
            </mx:series>
            <mx:horizontalAxis>
                <mx:CategoryAxis categoryField="Month"/>
            </mx:horizontalAxis>
        </mx:ColumnChart>

        <mx:Legend dataProvider="{myChart}" width="200"/>

        <mx:HBox>
            <mx:Form>
                <mx:FormItem label="x">
                    <mx:TextInput id="ti1" text="0"/>
                </mx:FormItem>
                <mx:FormItem label="y">
                    <mx:TextInput id="ti2" text="0"/>
                </mx:FormItem>
            </mx:Form>
            <mx:Form>
                <mx:FormItem label="Height">
                    <mx:TextInput id="ti3" text="0"/>
                </mx:FormItem>
                <mx:FormItem label="Width">
                    <mx:TextInput id="ti4" text="0"/>
                </mx:FormItem>
            </mx:Form>
        </mx:HBox>
        <mx:Button label="Get Items" click="getItems(event)" />
        <mx:TextArea id="ta1" height="100" width="300"/>
    </mx:Panel>
```

```
</mx:Application>
```

## Working with ChartItem objects to access chart data

To work with the data of a ChartItem, you typically cast it to its specific series type. For example, in a ColumnChart control, the ChartItem objects are of type ColumnSeriesItem. Doing this lets you access series-specific properties such as the yValue, xValue, and index of the ChartItem. You can also then access the data provider's object by using the `item` property. Finally, you can access the properties of the series by casting the `element` property to a series object.

Assuming that `a[i]` is a reference to a ChartItem object from an Array:

**To access the properties of the series item:**

```
var csi:ColumnSeriesItem = ColumnSeriesItem(a[i]);
trace(csi.yValue);
```

**To access the item's fields:**

```
var csi:ColumnSeriesItem = ColumnSeriesItem(a[i]);
trace(csi.item.Profit);
```

**To access the series properties:**

```
var csi:ColumnSeriesItem = ColumnSeriesItem(a[i]);
var cs:ColumnSeries = ColumnSeries(csi.element);
trace(cs.displayName);
```

## About selection events

When a user selects a chart item using mouse, keyboard or region selection, the chart's `change` event is dispatched. When following properties are updated through user interaction, a `change` event is dispatched:

- `selectedChartItem` and `selectedChartItems` (on the chart)
- `selectedItem`, `selectedItems`, `selectedIndex`, and `selectedIndices` (on the chart's series)

The `change` event does not contain references to the HitData or HitSet of the selected chart items. To access that information, you can use the chart's `selectedChartItem` and `selectedChartItems` properties.

When you select an item programmatically, no `change` event is dispatched. When you change the `selectedState` property of a chart item, no `change` event is dispatched.

## Clearing selections

You can clear all selected data points by using the chart control's `clearChartSelection()` method. The following example clears all selected data points when the user presses the ESC key:

```
<?xml version="1.0" ?>
<!--  charts/ClearItemSelection.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
    <mx:Script>
    <![CDATA[
        import mx.collections.ArrayCollection;
        import flash.events.KeyboardEvent;

        [Bindable]
        private var expensesAC:ArrayCollection = new ArrayCollection( [
            { Month: "Jan", Profit: 2000, Expenses: 1500, Amount: 450 },
            { Month: "Feb", Profit: 1000, Expenses: 200, Amount: 600 },
            { Month: "Mar", Profit: 1500, Expenses: 500, Amount: 300 },
            { Month: "Apr", Profit: 1800, Expenses: 1200, Amount: 900 },
            { Month: "May", Profit: 2400, Expenses: 575, Amount: 500 } ]);

        private function initApp():void {
            application.addEventListener(KeyboardEvent.KEY_UP, keyHandler);
        }

       // Clears all chart items selected when the user presses the ESC key.
        private function keyHandler(event:KeyboardEvent):void {
            var curKeyCode:int = event.keyCode;
            if (curKeyCode == 27) { // 27 is the keycode value for ESC
                myChart.clearChartSelection();
            }
        }
    ]]>
    </mx:Script>
    <mx:Panel height="100%" width="100%">
        <mx:PlotChart id="myChart"
            height="207"
            width="350"
            showDataTips="true"
            dataProvider="{expensesAC}"
            selectionMode="multiple"
        >
            <mx:series>
                <mx:PlotSeries id="series1"
                    xField="Expenses"
                    yField="Profit"
                    displayName="Expenses/Profit"
                    selectable="true"
                />
                <mx:PlotSeries id="series2"
                    xField="Amount"
                    yField="Expenses"
                    displayName="Amount/Expenses"
                    selectable="true"
                />
                <mx:PlotSeries id="series3"
                    xField="Profit"
                    yField="Amount"
                    displayName="Profit/Amount"
                    selectable="true"
                />
            </mx:series>
        </mx:PlotChart>
        <mx:Legend dataProvider="{myChart}" width="200"/>
    </mx:Panel>
</mx:Application>
```

In addition to clearing item selections programmatically, users can use the mouse or keyboard to clear items. If a user clicks anywhere on the chart control's background, and not over a data point, they clear all selections. If a user uses the up and down arrow keys to select a data point, they clear the existing data points. For more information, see "Keyboard and mouse selection" on page 1934.

## Using the selection API to create new charts

You can use the selection API to get some or all of the ChartItem objects of one chart, and then create a new chart with them. To do this, you create a new data provider for the new chart. To do this, you can call the ArrayCollection's `getItemAt()` method on the original chart's data provider and pass to it the original series' selected indices. This method then returns an object whose values you then add to an object in the new data provider.

The `selectedIndex` and `selectedIndices` properties of a chart's series represent the position of the chart item in the series. This position is also equivalent to the position of the chart item's underlying data object in the data provider.

The following example creates a PieChart control from the selected columns in the ColumnChart control. The PieChart control also allows selection; you can explode a piece by selecting it.

```
<?xml version="1.0" ?>
<!--  charts/MakeChartFromSelection.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
    <mx:Script>
    <![CDATA[
        import mx.collections.ArrayCollection;
        import mx.charts.chartClasses.ChartBase;
        import mx.charts.ChartItem;
        import mx.charts.series.items.ColumnSeriesItem;
        import mx.charts.PieChart;
        import mx.charts.series.PieSeries;
        import mx.charts.events.ChartItemEvent;

        [Bindable]
        public var newDataProviderAC:ArrayCollection;

        [Bindable]
        private var expensesAC:ArrayCollection = new ArrayCollection([
            { Month: "Jan", Expenses: 1500 },
            { Month: "Feb", Expenses: 200 },
            { Month: "Mar", Expenses: 500 },
            { Month: "Apr", Expenses: 1200 },
            { Month: "May", Expenses: 575 } ]);

        private function initApp():void {
            myColumnChart.addEventListener(ChartItemEvent.CHANGE,
                createNewChart);
            setupPieChart();
        }

        private function getNewDataProvider():ArrayCollection {
            newDataProviderAC = new ArrayCollection();

            for (var i:int=0; i<series1.selectedItems.length; i++) {
                var o:Object = new Object();
                o.Month =
                    expensesAC.getItemAt(series1.selectedIndices[i]).Month;
                o.Expenses =

expensesAC.getItemAt(series1.selectedIndices[i]).Expenses;
                newDataProviderAC.addItem(o);
            }
            return newDataProviderAC;
        }
        private var newChart:PieChart;
        private var newSeries:PieSeries;

        [Bindable]
        private var explodedPiece:Array;

        private function explodePiece(e:Event):void {
            explodedPiece = new Array();
            explodedPiece[newSeries.selectedIndex] = .2;
            newSeries.perWedgeExplodeRadius = explodedPiece;
        }

        private function setupPieChart():void {
            newChart  = new PieChart();
            newChart.showDataTips = true;
            newChart.selectionMode = "single";

            newSeries = new PieSeries();
            newSeries.field = "Expenses";
            newSeries.nameField = "Month";
            newSeries.setStyle("labelPosition", "callout");
            newSeries.setStyle("showDataEffect", "interpol");


            var newSeriesArray:Array = new Array();
            newSeriesArray.push(newSeries);
            newChart.series = newSeriesArray;

            newChart.addEventListener(ChartItemEvent.CHANGE, explodePiece);

            p1.addChild(newChart);
        }

        private function createNewChart(e:Event):void {
            newChart.dataProvider = getNewDataProvider();
        }
    ]]>
    </mx:Script>

    <mx:SeriesInterpolate id="interpol"
        duration="1000"
        elementOffset="0"
        minimumElementDuration="200"
    />

    <mx:Panel height="100%" width="100%" id="p1">
        <mx:ColumnChart id="myColumnChart"
            height="207"
            width="350"
            showDataTips="true"
            dataProvider="{expensesAC}"
            selectionMode="multiple"
        >
            <mx:series>
                <mx:ColumnSeries id="series1"
                    yField="Expenses"
                    displayName="Expenses"
                    selectable="true"
                />
            </mx:series>
            <mx:horizontalAxis>
                <mx:CategoryAxis categoryField="Month"/>
            </mx:horizontalAxis>
        </mx:ColumnChart>
    </mx:Panel>
</mx:Application>
```

## Dragging and dropping ChartItem objects

As an extension of the selection API, you can drag and drop chart items from one chart to another (or from a chart to some other object altogether).

To use drag and drop operations in your chart controls:

- Make the source chart drag enabled by setting it's `dragEnabled` property to `true`.
- Make the target chart drop enabled by setting it's `dropEnabled` property to `true`.
- Add event listeners for the `dragEnter` and `dragDrop` events on the target chart.
- In the `dragEnter` event handler, get a reference to the target chart and register it with the DragManager. To be a drop target, a chart must define an event handler for this event. You must call the `DragManager.acceptDragDrop()` method for the drop target to receive the drag and drop events.
- In the `dragDrop` event handler, get an Array of chart items that are being dragged and add that Array to the target chart's data provider. You do this by using the `event.dragSource.dataForFormat()` method. You must specify the String `"chartitems"` as the argument to the `dataForFormat()` method. This is because the chart-based controls have predefined values for the data format of drag data. For all chart controls, the format String is `"chartitems"`.

The following example lets you drag chart items from the ColumnChart control to the PieChart control. When the application starts, there is no PieChart control, but it becomes visible when the first chart item is dragged onto it. This example also examines the dragged chart items to ensure that they are not added more than once to the target chart. It does this by using the target data provider's `contains()` method.

```
<?xml version="1.0" ?>
<!--  charts/MakeChartFromDragDrop.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
    <mx:Style>
    </mx:Style>

    <mx:Script>
    <![CDATA[
        import mx.collections.ArrayCollection;

        import mx.charts.chartClasses.ChartBase;
        import mx.charts.ChartItem;
        import mx.charts.PieChart;
        import mx.charts.series.PieSeries;
        import mx.charts.events.ChartItemEvent;

        import mx.events.DragEvent;
        import mx.controls.List;
        import mx.managers.DragManager;
        import mx.core.DragSource;

        [Bindable]
        public var newDataProviderAC:ArrayCollection;

        [Bindable]
        private var expensesAC:ArrayCollection = new ArrayCollection([
            { Month: "Jan", Expenses: 1500 },
            { Month: "Feb", Expenses: 200 },
            { Month: "Mar", Expenses: 500 },
            { Month: "Apr", Expenses: 1200 },
            { Month: "May", Expenses: 575 } ]);

        private function initApp():void {
            setupPieChart();
        }

        private var newChart:PieChart;
        private var newSeries:PieSeries;

        [Bindable]
        private var explodedPiece:Array;
        private function explodePiece(e:Event):void {
            explodedPiece = new Array();
            explodedPiece[newSeries.selectedIndex] = .2;
            newSeries.perWedgeExplodeRadius = explodedPiece;
        }

        private function setupPieChart():void {
            newChart  = new PieChart();
            newChart.showDataTips = true;
            newChart.selectionMode = "multiple";
            newChart.dropEnabled= true;
            newChart.dragEnabled= false;

            newChart.height = 350;
            newChart.width = 350;

            newChart.addEventListener("dragEnter", doDragEnter);
            newChart.addEventListener("dragDrop", doDragDrop);

            newChart.dataProvider = newDataProviderAC;

            newSeries = new PieSeries();
            newSeries.field = "Expenses";
            newSeries.nameField = "Month";
            newSeries.setStyle("labelPosition", "callout");
            newSeries.setStyle("showDataEffect", "interpol");

            var newSeriesArray:Array = new Array();
            newSeriesArray.push(newSeries);
            newChart.series = newSeriesArray;

            newChart.addEventListener(ChartItemEvent.CHANGE, explodePiece);

            p2.addChild(newChart);
        }

        private function doDragEnter(event:DragEvent):void {
            // Get a reference to the target chart.
            var dragTarget:ChartBase = ChartBase(event.currentTarget);

            // Register the target chart with the DragManager.
            DragManager.acceptDragDrop(dragTarget);
        }

        private function doDragDrop(event:DragEvent):void {
            // Get a reference to the target chart.
            var dropTarget:ChartBase=ChartBase(event.currentTarget);

            // Get the dragged items from the drag initiator. When getting
            // items from chart controls, you must use the 'chartitems'
```

```
            // format.
            var items:Array = event.dragSource.dataForFormat("chartitems")
                as Array;

            // Trace status messages.
            trace("length: " + String(items.length));
            trace("format: " + String(event.dragSource.formats[0]));

            // Add each item to the drop target's data provider.
            for(var i:uint=0; i < items.length; i++) {

                // If the target data provider already contains the
                // item, then do nothing.
                if (dropTarget.dataProvider.contains(items[i].item)) {

                // If the target data provider does NOT already
                // contain the item, then add it.
                } else {
               dropTarget.dataProvider.addItem(items[i].item);
                }
            }
        }
    ]]>
    </mx:Script>

    <mx:SeriesInterpolate id="interpol"
        duration="1000"
        elementOffset="0"
        minimumElementDuration="200"
    />

    <mx:Panel id="p1" title="Source Chart" height="250" width="400">
        <mx:ColumnChart id="myColumnChart"
            height="207"
            width="350"
            showDataTips="true"
            dataProvider="{expensesAC}"
            selectionMode="multiple"
            dragEnabled="true"
            dropEnabled="false"
        >
            <mx:series>
                <mx:ColumnSeries id="series1"
                    yField="Expenses"
                    displayName="Expenses"
                    selectable="true"
                />
            </mx:series>
            <mx:horizontalAxis>
                <mx:CategoryAxis categoryField="Month"/>
            </mx:horizontalAxis>
        </mx:ColumnChart>
    </mx:Panel>

    <!-- This will be the parent of the soon-to-be created chart. -->
    <mx:Panel id="p2" title="Target Chart" height="400" width="400">
    </mx:Panel>
</mx:Application>
```

Chart controls also support the standard drag and drop methods of List-based controls such as `hideDropFeedback()` and `showDropFeedback()`. These methods display indicators under the mouse pointer to indicate whether drag and drop operations are allowed and where the items will be dropped.

For more information about drag and drop operations, see Chapter 31, “Using Drag and Drop,” on page 1195.

## Dropping ChartItem objects onto components

The target of a drag and drop operation from a chart control does not need to be another chart control. Instead, you can drag an object onto any Flex component using simple drag and drop rules. The following example lets you drag a column from the chart onto the TextArea. The TextArea then extracts data from the dropped item and displays that information in text.

```
<?xml version="1.0" ?>
<!--  charts/DragDropToComponent.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
    <![CDATA[
        import mx.collections.ArrayCollection;

        import mx.events.DragEvent;
        import mx.controls.List;
        import mx.managers.DragManager;
        import mx.core.DragSource;

        import mx.charts.chartClasses.ChartBase;
        import mx.charts.ChartItem;
        import mx.charts.events.ChartItemEvent;
        import mx.charts.series.items.ColumnSeriesItem;

        [Bindable]
        private var clothing_sales:ArrayCollection = new ArrayCollection( [
            { Month: "Jan", Pants: 35, Shirts:39, Fedoras: 29 },
            { Month: "Feb", Pants: 32, Shirts:17, Fedoras: 14 },
            { Month: "Mar", Pants: 27, Shirts:27, Fedoras: 38 } ]);

        private function doDragEnter(event:DragEvent):void {
            var dragTarget:TextArea = TextArea(event.currentTarget);
            DragManager.acceptDragDrop(dragTarget);
        }

        private function doDragDrop(event:DragEvent):void {
            var dropTarget:TextArea = TextArea(event.currentTarget);

            var curItem:ColumnSeriesItem =

ColumnSeriesItem(event.dragSource.dataForFormat("chartitems")[0]);
            var curSeries:ColumnSeries = ColumnSeries(curItem.element);

            var clothingType:String = curSeries.displayName;
            var numSold:String = curItem.yValue.toString();
            var monthSold:String = curItem.item.Month;

            ta1.text = "You sold " + numSold + " " +
                clothingType + " in " + monthSold + ".";
        }
    ]]>
    </mx:Script>
        <mx:Panel title="Rearrange Items in ColumnChart">
            <mx:ColumnChart id="myChart"
                height="225"
                showDataTips="true"
                dataProvider="{clothing_sales}"
                selectionMode="single"
                dragEnabled="true"
            >
                <mx:horizontalAxis>
                    <mx:CategoryAxis categoryField="Month"/>
                </mx:horizontalAxis>
                <mx:series>
                    <mx:ColumnSeries id="columnSeries1"
                        xField="Month"
                        yField="Pants"
                        displayName="Pants"
                        selectable="true"
                    />
                    <mx:ColumnSeries id="columnSeries2"
                        xField="Month"
                        yField="Shirts"
                        displayName="Shirts"
                        selectable="true"
                    />
                    <mx:ColumnSeries id="columnSeries3"
                        xField="Month"
                        yField="Fedoras"
                        displayName="Fedoras"
                        selectable="true"
                    />
                </mx:series>
            </mx:ColumnChart>
            <mx:HBox>
                <mx:Legend dataProvider="{myChart}"/>
                <mx:TextArea id="ta1"
                    height="75"
                    width="200"
                    dragEnter="doDragEnter(event)"
                    dragDrop="doDragDrop(event)"
                />
            </mx:HBox>
        </mx:Panel>
</mx:Application>
```

## Changing the drag image

When you drag a chart item off of a source chart, an outline of the underlyig chart item appears as the drag image. For example, if you drag a column from a ColumnChart control, a column appears under the mouse pointer to represent the item that is being dragged.

You can customize the image that is displayed during a drag operation by using the overriding the default defintion of the `dragImage` property in a custom chart class. You do this by embedding your new image (or defining it in ActionScript), overriding the `dragImage()` getter method, and returning the new image.

The default location, in coordinates, of the drag image is 0,0 unless you override the DragManager's `doDrag()` method. You can also set the starting location of the drag proxy image by using the x and y coordinates of the image proxy in the `dragImage()` getter.

The following example custom chart class embeds an image and returns it in the `dragImage()` getter method. This example also positions the drag image proxy so that the mouse pointer is near its lower right corner.

```
// charts/MyColumnChart.as
package {
    import mx.charts.ColumnChart;
    import mx.core.IUIComponent;
    import mx.controls.Image;

    public class MyColumnChart extends ColumnChart {
        [Embed(source="../../images/charting/dollarSign.png")]
        public var dollarSign:Class;

        public function MyColumnChart() {
            super();
        }

        override protected function get dragImage():IUIComponent {
             var imageProxy:Image = new Image();
             imageProxy.source = dollarSign;

             var imageHeight:Number = 50;
             var imageWidth:Number = 32;

             imageProxy.height = imageHeight;
             imageProxy.width = imageWidth;

             // Position the image proxy above and to the left of
             // the mouse pointer.
             imageProxy.x = this.mouseX - imageWidth;
             imageProxy.y = this.mouseY - imageHeight;

             return imageProxy;
        }

    }
}
```

The following example uses the MyColumnChart custom chart class to define its custom drag image:

```
<?xml version="1.0" ?>
<!--  charts/MakeChartFromDragDrop.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()" xmlns:local="*">
    <mx:Style>
    </mx:Style>

    <mx:Script>
    <![CDATA[
        import mx.collections.ArrayCollection;

        import mx.charts.chartClasses.ChartBase;
        import mx.charts.ChartItem;
        import mx.charts.PieChart;
        import mx.charts.series.PieSeries;
        import mx.charts.events.ChartItemEvent;

        import mx.events.DragEvent;
        import mx.controls.List;
        import mx.managers.DragManager;
        import mx.core.DragSource;

        import mx.controls.Image;
        import flash.events.MouseEvent;

        [Bindable]
        public var newDataProviderAC:ArrayCollection;

        [Bindable]
        private var expensesAC:ArrayCollection = new ArrayCollection([
            { Month: "Jan", Expenses: 1500 },
            { Month: "Feb", Expenses: 200 },
            { Month: "Mar", Expenses: 500 },
            { Month: "Apr", Expenses: 1200 },
            { Month: "May", Expenses: 575 } ]);

        private function initApp():void {
            setupPieChart();
        }

        private var newChart:PieChart;
        private var newSeries:PieSeries;

        [Bindable]
        private var explodedPiece:Array;

        private function explodePiece(e:Event):void {
```

```
        explodedPiece = new Array();
        explodedPiece[newSeries.selectedIndex] = .2;
        newSeries.perWedgeExplodeRadius = explodedPiece;
    }

    private function setupPieChart():void {
        newChart  = new PieChart();
        newChart.showDataTips = true;
        newChart.selectionMode = "multiple";
        newChart.dropEnabled= true;
        newChart.dragEnabled= false;

        newChart.height = 350;
        newChart.width = 350;

        newChart.addEventListener("dragEnter", doDragEnter);
        newChart.addEventListener("dragDrop", doDragDrop);

        newChart.dataProvider = newDataProviderAC;

        newSeries = new PieSeries();
        newSeries.field = "Expenses";
        newSeries.nameField = "Month";
        newSeries.setStyle("labelPosition", "callout");
        newSeries.setStyle("showDataEffect", "interpol");

        var newSeriesArray:Array = new Array();
        newSeriesArray.push(newSeries);
        newChart.series = newSeriesArray;

        newChart.addEventListener(ChartItemEvent.CHANGE, explodePiece);

        p2.addChild(newChart);
    }

    private function doDragEnter(event:DragEvent):void {
        var dragTarget:ChartBase = ChartBase(event.currentTarget);
        DragManager.acceptDragDrop(dragTarget);
    }

    private function doDragDrop(event:DragEvent):void {
        var dropTarget:ChartBase=ChartBase(event.currentTarget);

        var items:Array = event.dragSource.dataForFormat("chartitems")
            as Array;

        for(var i:uint=0; i < items.length; i++) {
            if (dropTarget.dataProvider.contains(items[i].item)) {
            } else {
           dropTarget.dataProvider.addItem(items[i].item);
```

```
                }
            }
        }
    ]]>
    </mx:Script>

    <mx:SeriesInterpolate id="interpol"
        duration="1000"
        elementOffset="0"
        minimumElementDuration="200"
    />

    <mx:Panel id="p1" title="Source Chart" height="250" width="400">
        <local:MyColumnChart id="myChart"
            height="207"
            width="350"
            showDataTips="true"
            dataProvider="{expensesAC}"
            selectionMode="multiple"
            dragEnabled="true"
            dropEnabled="false"
        >
            <local:series>
                <mx:ColumnSeries id="series1"
                    yField="Expenses"
                    displayName="Expenses"
                    selectable="true"
                />
            </local:series>
            <local:horizontalAxis>
                <mx:CategoryAxis categoryField="Month"/>
            </local:horizontalAxis>
        </local:MyColumnChart>
    </mx:Panel>

    <mx:Panel id="p2" title="Target Chart" height="400" width="400">
    </mx:Panel>
</mx:Application>
```

## Drawing on chart controls

Flex includes the ability to add graphical elements to your chart controls such as lines, boxes, and elipses. Graphical children of chart controls can also include any control that is a subclass of UIComponent, including list boxes, combo boxes, labels, and even other chart controls.

When working with chart controls, Flex converts x and y coordinates into data coordinates. Data coordinates define locations relative to the data in the chart's underlying data provider. This lets you position graphical elements relative to the location of data points in the chart without having to first convert their positions to x and y coordinates. For example, you can draw a line from the top of a column in a ColumnChart to the the top of another column, showing a trend line.

To add data graphics to a chart control, you use the mx.charts.chartClasses.CartesianDataCanvas (for Cartesian charts). You can either attach visual controls to the canvas or you can draw on the canvas using drawing methods.

You attach visual controls to the data canvases in the same way that you programmatically add controls to an application: you add them as child controls. In this case, the method you call to add children is `addDataChild()`, which lets you add any DisplayObject instance to the canvas. This method lets you take advantage of the data coordinates that the canvas uses. Like most containers, you can also use the `addChild()` and `addChildAt()` methods. With these methods, you can then adjust the location of the DisplayObject to data coordinates by using the canvas's `updateDataChild()` method.

To draw on the data canvases, you use drawing methods that are similar to the methods of the flash.display.Graphics class. The canvases define a set of methods that you can use to create vector shapes such as circles, squares, lines, and other shapes. These methods include the line-drawing methods such as `lineTo()`, `moveTo()`, and `curveTo()`, as well as the fill methods such as `beginFill()`, `beginGradientFill()`, `beginBitmapFill()`, and `endFill()`. Convenience methods such as `drawRect()`, `drawRoundedRect()`, `drawEllipse()`, and `drawCircle()` are also available on the data canvases.

All drawn graphics such as lines and shapes remain visible on the data canvas until you call the canvas's `clear()` method. To remove child objects, you can use the `removeChild()` and `removeChildAt()` methods.

A canvas can either be in the foreground (in front of the data points) or in the background (behind the data points). To add a canvas to the foreground, you add it to the chart's `annotationElement` Array. To add a canvas to the background, you add it to the chart's `backgroundElements` Array.

The data canvases have the following limitations:

- There is no drag and drop support for children of the data canvases
- You cannot use the chart selection APIs to select objects on the data canvases

The following example adds a CartesianDataCanvas as an annotation element to the ColumnChart control. It uses the graphics methods to draw a line between the columns that you select.

```
<?xml version="1.0"?>
<!-- charts/DrawLineBetweenSelectedItems.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
    import mx.collections.ArrayCollection;
    import mx.charts.series.items.ColumnSeriesItem;
    import mx.charts.ChartItem;

     [Bindable]
     public var profits:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:1300},
        {Month:"Feb", Profit:750},
        {Month:"Mar", Profit:1100},
        {Month:"Apr", Profit:1000},
        {Month:"May", Profit:980},
        {Month:"Jun", Profit:1500},
        {Month:"Jul", Profit:2060},
        {Month:"Aug", Profit:1700},
        {Month:"Sep", Profit:1690},
        {Month:"Oct", Profit:2200},
        {Month:"Nov", Profit:2550},
        {Month:"Dec", Profit:3000}
     ]);

     private function connectTwoPoints(month1:String,
        value1:Number,
        month2:String,
        value2:Number
     ):void {
        canvas.clear();
        canvas.lineStyle(4,
            0xCCCCCC,
            .75,
            true,
            LineScaleMode.NORMAL,
            CapsStyle.ROUND,
            JointStyle.MITER,
            2
        );
        canvas.moveTo(month1, value1);
        canvas.lineTo(month2, value2);

        l1.text = "Month: " + month1;
        l2.text = "Profit: " + value1;
        l3.text = "Month: " + month2;
        l4.text = "Profit: " + value2;
        chartHasLine = true;
     }

    private var s1:String = new String();
    private var s2:String = new String();
    private var v1:Number = new Number();
    private var v2:Number = new Number();

    // Set this to true initially so that the chart doesn't
    // draw a line when the first item is clicked.
    private var chartHasLine:Boolean = true;

    private function handleChange(event:Event):void {

        var sci:ColumnSeriesItem =
            ColumnSeriesItem(myChart.selectedChartItem);

        if (chartHasLine) {
            canvas.clear();

            s1 = sci.item.Month;
            v1 = sci.item.Profit;

            chartHasLine = false;
        } else {
            s2 = sci.item.Month;
            v2 = sci.item.Profit;

            connectTwoPoints(s1, v1, s2, v2);
        }
    }
  ]]></mx:Script>
  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart"
        dataProvider="{profits}"
        selectionMode="single"
        change="handleChange(event)"
     >
        <mx:annotationElements>
            <mx:CartesianDataCanvas id="canvas" includeInRanges="true"/>
        </mx:annotationElements>

        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{profits}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
```

```
        <mx:series>
           <mx:ColumnSeries
                id="series1"
                xField="Month"
                yField="Profit"
                displayName="Profit"
                selectable="true"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>

     <mx:VBox>
        <mx:Button id="b1"
            label="Connect Two Points"
            click="connectTwoPoints('Jan', 1300, 'Dec', 3000);"
        />
        <mx:Button id="b2" click="canvas.clear()" label="Clear Line"/>
     </mx:VBox>

     <mx:HBox>
         <mx:VBox>
             <mx:Label text="First Item"/>
             <mx:Label id="l1"/>
             <mx:Label id="l2"/>
         </mx:VBox>

         <mx:VBox>
             <mx:Label text="Second Item"/>
             <mx:Label id="l3"/>
             <mx:Label id="l4"/>
         </mx:VBox>
     </mx:HBox>
   </mx:Panel>
</mx:Application>
```

The following example uses the `addDataChild()` method to add children to the data canvas. It adds labels to each of the columns that you select in the ColumnChart control.

```
<?xml version="1.0"?>
<!-- charts/AddLabelsWithLines.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
    import mx.collections.ArrayCollection;
    import mx.charts.series.items.ColumnSeriesItem;
    import mx.charts.ChartItem;

     [Bindable]
     public var profits:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:1300},
        {Month:"Feb", Profit:750},
        {Month:"Mar", Profit:1100},
        {Month:"Apr", Profit:1000},
        {Month:"May", Profit:980},
        {Month:"Jun", Profit:1500},
        {Month:"Jul", Profit:2060},
        {Month:"Aug", Profit:1700},
        {Month:"Sep", Profit:1690},
        {Month:"Oct", Profit:2200},
        {Month:"Nov", Profit:2550},
        {Month:"Dec", Profit:3000}
     ]);

     private function connectTwoPoints(month1:String,
        value1:Number,
        month2:String,
        value2:Number
     ):void {
        canvas.clear();
        canvas.lineStyle(4,
            0xCCCCCC,
            .75,
            true,
            LineScaleMode.NORMAL,
            CapsStyle.ROUND,
            JointStyle.MITER,
            2
        );
        canvas.moveTo(month1, value1);
        canvas.lineTo(month2, value2);

        l1.text = "Month: " + month1;
        l2.text = "Profit: " + value1;
        l3.text = "Month: " + month2;
        l4.text = "Profit: " + value2;
        chartHasLine = true;
     }

    private var s1:String = new String();
    private var s2:String = new String();
    private var v1:Number = new Number();
    private var v2:Number = new Number();

    // Set this to true initially so that the chart doesn't
    // draw a line when the first item is clicked.
    private var chartHasLine:Boolean = true;

    private function handleChange(event:Event):void {

        var sci:ColumnSeriesItem =
            ColumnSeriesItem(myChart.selectedChartItem);

        if (chartHasLine) {
            canvas.clear();

            s1 = sci.item.Month;
            v1 = sci.item.Profit;

            addLabelsToColumn(s1,v1);

            chartHasLine = false;
        } else {
            s2 = sci.item.Month;
            v2 = sci.item.Profit;

            addLabelsToColumn(s2,v2);

            connectTwoPoints(s1, v1, s2, v2);
        }
    }

    [Bindable]
    public var columnLabel:Label;

    private function addLabelsToColumn(s:String, n:Number):void {
        columnLabel = new Label();
        columnLabel.setStyle("fontWeight", "bold");
        columnLabel.setStyle("color", "0x660000");
        columnLabel.text = s + ": " + "$" + n;

        // This adds any DisplayObject as child to current canvas.
        canvas.addDataChild(columnLabel, s, n);
    }

  ]]></mx:Script>
  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart"
        dataProvider="{profits}"
        selectionMode="single"
        change="handleChange(event)"
     >
        <mx:annotationElements>
            <mx:CartesianDataCanvas id="canvas" includeInRanges="true"/>
        </mx:annotationElements>

        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{profits}"
                categoryField="Month"
           />
        </mx:horizontalAxis>

        <mx:series>
           <mx:ColumnSeries
                id="series1"
                xField="Month"
                yField="Profit"
                displayName="Profit"
                selectable="true"
           />
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>

     <mx:Button id="b1"
        label="Connect Two Points"
        click="connectTwoPoints('Jan', 1300, 'Dec', 3000);"/>

     <mx:HBox>
         <mx:VBox>
             <mx:Label text="First Item"/>
             <mx:Label id="l1"/>
             <mx:Label id="l2"/>
         </mx:VBox>

         <mx:VBox>
             <mx:Label text="Second Item"/>
             <mx:Label id="l3"/>
             <mx:Label id="l4"/>
         </mx:VBox>
     </mx:HBox>

  </mx:Panel>
</mx:Application>
```

## Using offsets for data coordinates

The data canvas classes also let you add offsets to the position of data graphics. You do this by defining the data coordinates with the CartesianCanvasValue constructor rather than passing a data coordinate to the drawing or `addDataChild()` methods. When you define a data coordinate with the CartesianCanvasValue, you pass the data coordinate as the first argument, but you can pass an offset as the second argument.

The following example lets you specify an offset with an HSlider control. This offset is used when adding a data child to the canvas.

```
<?xml version="1.0"?>
<!-- charts/AddLabelsWithOffsetLines.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
    import mx.collections.ArrayCollection;
    import mx.charts.series.items.ColumnSeriesItem;
    import mx.charts.ChartItem;
    import mx.charts.chartClasses.CartesianCanvasValue;

     [Bindable]
     public var profits:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:1300},
        {Month:"Feb", Profit:750},
        {Month:"Mar", Profit:1100},
        {Month:"Apr", Profit:1000},
        {Month:"May", Profit:980},
        {Month:"Jun", Profit:1500},
        {Month:"Jul", Profit:2060},
        {Month:"Aug", Profit:1700},
        {Month:"Sep", Profit:1690},
        {Month:"Oct", Profit:2200},
        {Month:"Nov", Profit:2550},
        {Month:"Dec", Profit:3000}
     ]);

     private function connectTwoPoints(month1:String,
        value1:Number,
        month2:String,
        value2:Number
     ):void {
        canvas.clear();
        canvas.lineStyle(4,
            0xCCCCCC,
            .75,
            true,
            LineScaleMode.NORMAL,
            CapsStyle.ROUND,
            JointStyle.MITER,
            2
        );
        canvas.moveTo(month1, value1);
        canvas.lineTo(month2, value2);

        l1.text = "Month: " + month1;
        l2.text = "Profit: " + value1;
        l3.text = "Month: " + month2;
        l4.text = "Profit: " + value2;
```

```
            chartHasLine = true;
         }

        private var s1:String = new String();
        private var s2:String = new String();
        private var v1:Number = new Number();
        private var v2:Number = new Number();

        // Set this to true initially so that the chart doesn't
        // draw a line when the first item is clicked.
        private var chartHasLine:Boolean = true;

        private function handleChange(event:Event):void {

            var sci:ColumnSeriesItem =
                ColumnSeriesItem(myChart.selectedChartItem);

            if (chartHasLine) {
                canvas.clear();

                s1 = sci.item.Month;
                v1 = sci.item.Profit;

                addLabelsToColumn(s1,v1);

                chartHasLine = false;
            } else {
                s2 = sci.item.Month;
                v2 = sci.item.Profit;

                addLabelsToColumn(s2,v2);

                connectTwoPoints(s1, v1, s2, v2);
            }
        }

        [Bindable]
        public var labelOffset:Number = 0;

        [Bindable]
        public var columnLabel:Label;

        private function addLabelsToColumn(s:String, n:Number):void {
            columnLabel = new Label();
            columnLabel.setStyle("fontWeight", "bold");
            columnLabel.setStyle("color", "0x660000");
            columnLabel.text = s + ": " + "$" + n;

            // Use the CartesianCanvasValue constructor to specify
            // an offset for data coordinates.
            canvas.addDataChild(columnLabel,
                new CartesianCanvasValue(s, labelOffset),
                new CartesianCanvasValue(n, labelOffset)
            );
        }

      ]]></mx:Script>
      <mx:Panel title="Column Chart">
         <mx:ColumnChart id="myChart"
            dataProvider="{profits}"
            selectionMode="single"
            change="handleChange(event)"
         >
            <mx:annotationElements>
                <mx:CartesianDataCanvas id="canvas" includeInRanges="true"/>
            </mx:annotationElements>

            <mx:horizontalAxis>
               <mx:CategoryAxis
                    dataProvider="{profits}"
                    categoryField="Month"
               />
            </mx:horizontalAxis>

            <mx:series>
               <mx:ColumnSeries
                    id="series1"
                    xField="Month"
                    yField="Profit"
                    displayName="Profit"
                    selectable="true"
               />
            </mx:series>
         </mx:ColumnChart>
         <mx:Legend dataProvider="{myChart}"/>

         <mx:Button id="b1"
            label="Connect Two Points"
            click="connectTwoPoints('Jan', 1300, 'Dec', 3000);"
         />

         <mx:HBox>
             <mx:VBox>
                 <mx:Label text="First Item"/>
                 <mx:Label id="l1"/>
                 <mx:Label id="l2"/>
             </mx:VBox>

             <mx:VBox>
```

```
            <mx:Label text="Second Item"/>
            <mx:Label id="l3"/>
            <mx:Label id="l4"/>
        </mx:VBox>
     </mx:HBox>

    <mx:HSlider id="hSlider" minimum="-50" maximum="50" value="0"
        dataTipPlacement="top"
        tickColor="black"
        snapInterval="1" tickInterval="10"
        labels="['-50','0','50']"
        allowTrackClick="true"
        liveDragging="true"
        change="labelOffset=hSlider.value"
    />

  </mx:Panel>
</mx:Application>
```

## Using multiple axes with data canvases

A CartesianDataCanvas is specific to a certain data space, which is defined by the bounds of the axis. If no axis is specified then the canvas uses the primary axes of the chart as its bounds; If you have a chart with multiple axes, you can specify which axes the data canvas should use by setting the values of the horizontalAxis or verticalAxis properties:

```
<ColumnChart width="100%" height="100%" creationComplete="createLabel();">
  <annotationElements>
    <CartesianDataCanvas id="canvas"
      includeInRanges="true"
      horizontalAxis={h1}
      verticalAxis={v1}/>
  </annotationElements>
  ...
</ColumnChart>
```