2015. java学习交流群

8918 1289

每天有免费的Java学习课堂

——学习Java就是这么简单

——为Java而燃烧——

ADOBE® FLEX® 3

ADVANCED DATA VISUALIZATION DEVELOPER GUIDE

Adobe

© 2008 Adobe Systems Incorporated. All rights reserved.

Adobe® Flex® 3 Data Visualization Developer Guide

If this guide is distributed with software that includes an end-user agreement, this guide, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. Except as permitted by any such license, no part of this guide may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior written permission of Adobe Systems Incorporated. Please note that the content in this guide is protected under copyright law even if it is not distributed with software that includes an end-user license agreement.

The content of this guide is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this guide.

Please remember that existing artwork or images that you may want to include in your project may be protected under copyright law. The unauthorized incorporation of such material into your new work could be a violation of the rights of the copyright owner. Please be sure to obtain any permission required from the copyright owner.

Any references to company names in sample templates are for demonstration purposes only and are not intended to refer to any actual organization.

Adobe, the Adobe logo, Flash, and Flex, are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

Windows is either a registered trademark or trademarks of Microsoft Corporation in the United States and/or other countries. Java is a trademark or registered trademark of Sun Microsystems, Inc. in the United States and other countries. All other trademarks are the property of their respective owners.

This product includes software developed by the Apache Software Foundation (http://www.apache.org/).

This product contains either BISAFE and/or TIPEM software by RSA Data Security, Inc.

The Flex Builder 3 software contains code provided by the Eclipse Foundation ("Eclipse Code"). The source code for the Eclipse Code as contained in Flex Builder 3 software ("Eclipse Source Code") is made available under the terms of the Eclipse Public License v1.0 which is provided herein, and is also available at http://www.eclipse.org/legal/epl-v10.html.

Adobe Systems Incorporated, 345 Park Avenue, San Jose, CA 95110-2704, USA.

Notice to U.S. government end users. The software and documentation are "Commercial Items," as that term is defined at 48 C.F.R. §2.101, consisting of "Commercial Computer Software" and "Commercial Computer Software Documentation," as such terms are used in 48 C.F.R. §12.212 or 48 C.F.R. §227.7202, as applicable. Consistent with 48 C.F.R. §12.212 or 48 C.F.R. §§227.7202-1 through 227.7202-4, as applicable, the Commercial Computer Software and Commercial Computer Software Documentation are being licensed to U.S. Government end users (a) only as Commercial items and (b) with only those rights as are granted to all other end users pursuant to the terms and conditions herein. Unpublished-rights reserved under the copyright laws of the United States. For U.S. Government End Users, Adobe agrees to comply with all applicable equal opportunity laws including, if appropriate, the provisions of Executive Order 11246, as amended, Section 402 of the Vietnam Era Veterans Readjustment Assistance Act of 1974 (38 USC 4212), and Section 503 of the Rehabilitation Act of 1973, as amended, and the regulations at 41 CFR Parts 60-1 through 60-60, 60-250 ,and 60-741. The affirmative action clause and regulations contained in the preceding sentence shall be incorporated by reference.

Contents

Part 1: Flex Charting Components

Chapter 1: Introduction to Charts
About charting 2
Using the charting controls 3
About the series classes 8
About the axis classes 9
About charting events 10
Creating charts in ActionScript 10
Defining chart data 14

Chapter 2: Chart Types
Using area charts 36
Using bar charts 40
Using bubble charts 41
Using candlestick charts 46
Using column charts 50
Using HighLowOpenClose charts 53
Using line charts 56
Using pie charts 63
Using plot charts 70
Using multiple data series 73
Using multiple axes 75

Chapter 3: Formatting Charts
Applying chart styles 81
Setting padding properties 91
Formatting tick marks 95
Formatting axis lines 97
Using strokes with chart controls 99
Using fills with chart controls 104
Using filters with chart controls 116
Using chart grid lines 121
Positioning chart axes 127
Rotating chart axis labels 128
Skinning ChartItem objects 132
Formatting Legend controls 140

Chapter 4: Displaying Data and Labels
Working with axes 145
About the CategoryAxis class 145
About the NumericAxis class 147
Adding axis titles 159

Defining axis labels 164
Using data labels 171
Using DataTip objects 179
Using per-item fills 189
Using the minField property 195
Stacking charts 197
Using Legend controls 204

Chapter 5: Using Events and Effects in Charts
Handling user interactions with charts 210
Using effects with charts 221
Drilling down into data 235
Selecting chart items 240
Drawing on chart controls 264

Part 2: Advanced Data Grid Controls and Automation Tools

Chapter 6: Using the AdvancedDataGrid Control
About the AdvancedDataGrid control 275
Sorting by multiple columns 278
Styling rows and columns 280
Selecting multiple cells and rows 284
Hierarchical and grouped data display 287
Displaying hierarchical data 294
Displaying grouped data 296
Creating column groups 309
Using item renderers with the AdvancedDataGrid control 313
Keyboard navigation 320

Chapter 7: Creating OLAP Data Grids
About OLAP data grids 322
Creating an OLAP schema 330
Creating OLAP queries 333
Writing a query for a simple OLAP cube 338
Writing a query for a complex OLAP cube 344
Creating multidimensional axis in an OLAPDataGrid control 348
Configuring the display of an OLAPDataGrid 352

Chapter 8: Creating Applications for Testing
About automating applications with Flex 356
Tasks and techniques for testable applications overview 357
Compiling applications for testing 358
Creating testable applications 360
Understanding the automation framework 362
Instrumenting events 365
Instrumenting custom components 367
Instrumenting composite components 373

Example: Instrumenting the RandomWalk custom component for QTP 375

Chapter 9: Creating Custom Agents

About creating custom agents 381
About the automation APIs 382
Understanding the automation flow 385
Creating agents 387
Creating a recording agent 388

Part 1: Flex Charting Components

Topics

Introduction to Charts 2
Chart Types 36
Formatting Charts 81
Displaying Data and Labels 145
Using Events and Effects in Charts 210

Chapter 1: Introduction to Charts

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.

The charting controls are a feature of Adobe Flex® Builder™ Professional. You can create charts in your Flex applications with Flex Builder Standard, but the charting controls will have a watermark on them.

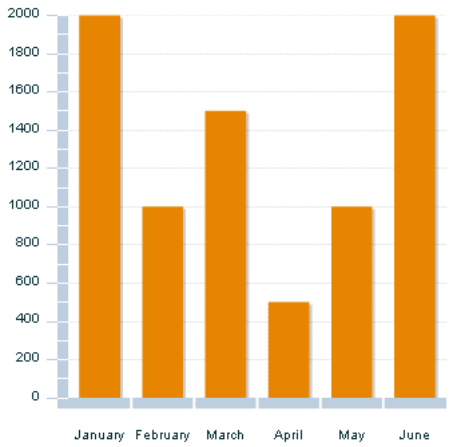
Topics

About charting 2
Using the charting controls 3
About the axis classes 9
About charting events 10
Creating charts in ActionScript 10
Defining chart data 15

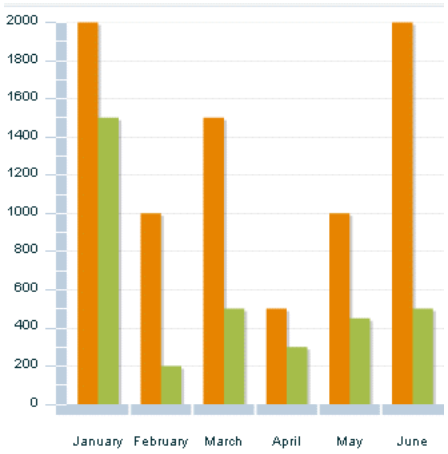
About charting

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.



Flex Builder Professional is required to use Flex charting controls without watermarks.

Using the charting controls

Flex charting controls 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:axis_type/>
    </mx:horizontalAxis>
    <mx:verticalAxis>
        <mx:axis_type/>
    </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.
	You can also define a second set of series for each chart, to show multiple data series in a single chart.
	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.
	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.
	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 15.

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}" showDataTips="true">
        <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="500" width="500">
     <mx:BarChart id="myChart"
        dataProvider="{expenses}"
        height="100%"
        width="100%"
        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>
```

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 chart 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 DataTip objects.

A BarChart typically specifies one or more 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 198.

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 73.

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 105.

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 116.

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 171.

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 196.

About the axis classes

Flex charting controls support the following types of axes:

CategoryAxis 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 2 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.

There are exceptions. 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. When you create PieChart controls, you also do not specify either axis, because PieChart controls use a single set of data points to draw wedges that represent a percentage of the whole. In PieChart controls, you define a `nameField` on the chart's data series, rather than a `categoryField` or `name` on the axes for labels and legends.

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 "Formatting Charts" on page 81.

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 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 orientation to calculate the labels that appear along the x-axis and y-axis of the chart. The labels of a column chart, for example, have the following default values:

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

The y-axis The minimum value on the y-axis is small enough for the chart data, and the maximum value is large enough based to accomodate the chart data.

For more information about chart axes, see "Working with axes" on page 145.

About charting events

The chart controls include events that accommodate user interaction with data points in charts. These events are described in "Using Events and Effects in Charts" on page 210.

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]
        private var profit04:ArrayCollection = new ArrayCollection([
            {Month: "Jan", Profit: 2000},
            {Month: "Feb", Profit: 1000},
            {Month: "Mar", Profit: 1500}
        ]);

        [Bindable]
        private var profit05:ArrayCollection = new ArrayCollection([
            {Month: "Jan", Profit: 2200},
            {Month: "Feb", Profit: 1200},
            {Month: "Mar", Profit: 1700}
        ]);

        [Bindable]
        private var profit06:ArrayCollection = new ArrayCollection([
            {Month: "Jan", Profit: 2400},
```

```
            {Month: "Feb", Profit: 1400},
            {Month: "Mar", Profit: 1900}
        ]);

        private var series1:ColumnSeries;
        private var series2:ColumnSeries;

        private 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;
            }
        }

        private function resetApp():void {
            myChart.series = [ series0 ];
            series1 = null;
            series2 = null;
        }
    ]]></mx:Script>

    <mx:Panel title="Column Chart">
        <mx:ColumnChart id="myChart" dataProvider="{profit04}" showDataTips="true">
            <mx:horizontalAxis>
                <mx:CategoryAxis categoryField="Month"/>
            </mx:horizontalAxis>
            <mx:series>
                <mx:ColumnSeries dataProvider="{profit04}"
                    id="series0"
                    yField="Profit"
                    xField="Month"
                    displayName="2004"
                />
            </mx:series>
        </mx:ColumnChart>
        <mx:Legend dataProvider="{myChart}"/>
    </mx:Panel>
    <mx:HBox>
        <mx:Button id="b1" label="Add More Series To Chart" click="addMoreSeries()"/>
        <mx:Button id="b2" label="Reset" click="resetApp()"/>
    </mx:HBox>

</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`.

The data provider can contain complex objects, or objects within objects. For example, a data provider object can have the following structure:

```
{month: "Aug", close: {High:45.87,Low:12.2}, open:25.19}
```

In this case, you cannot simply refer to the field of the data provider by using a `categoryField`, `xField`, or similar flat naming convention. Rather, you use the `dataFunction` of the series or axis to drill down into the data provider. For more information on working with complex data, see "Structure of chart data" on page 28.

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

1 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 use, see the data series entry in *Adobe Flex Language Reference*. For more information on data providers, see "Data provider controls" on page 227 in *Adobe Flex 3 Developer Guide*.

Sources 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.

There are some limitations on the structure of the chart data, and how to reference chart data if it is constructed with complex objects. For more information, see "Structure of chart data" on page 28.

For more information on data providers, see "Using Data Providers and Collections" on page 137 in *Adobe Flex 3 Developer Guide*.

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}" showDataTips="true">
        <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}" showDataTips="true">
      <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}" showDataTips="true">
      <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 19.

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}" showDataTips="true">
      <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 an object in MXML usig child tags rather than attributes:

```
<?xml version="1.0"?>
<!-- charts/ArrayCollectionOfVerboseMXMLObjects.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}" showDataTips="true">
      <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}" showDataTips="true">
       <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}" showDataTips="true">
        <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}" showDataTips="true">
        <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 Adobe® Flash® Player or Adobe® AIR™ 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 or AIR started.
     o.time = getTimer();

     // Get the total memory Flash Player or AIR 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 controls 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 "Using Data Providers and Collections" on page 137 in *Adobe Flex 3 Developer Guide*.

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="myChart" dataProvider="{results.result}" showDataTips="true">
        <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:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

You can use more complex XML to define the data provider's data. For example, an XML-based data provider can have nested tags. In that case, however, you must use a `dataFunction` to define the fields that the chart uses. For more information, see "Structure of chart data" on page 28.

To use an ArrayCollection as the chart's data provider, you convert the Model to an ArrayCollection, 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="myChart" dataProvider="{myAC}" showDataTips="true">
        <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:Legend dataProvider="{myChart}"/>
  </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="myChart"
        dataProvider="{srv.lastResult.data.result}"
        showDataTips="true"
     >
        <mx:horizontalAxis>
           <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries yField="apple" displayName="Apple" name="Apple"/>
           <mx:LineSeries yField="orange" displayName="Orange" name="Orange"/>
           <mx:LineSeries yField="banana" displayName="Banana" name="Banana"/>
        </mx:series>
     </mx:LineChart>
     <mx:Legend dataProvider="{myChart}"/>
  </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="myChart" dataProvider="{myData}" showDataTips="true">
         <mx:horizontalAxis>
            <mx:CategoryAxis categoryField="month"/>
         </mx:horizontalAxis>
         <mx:series>
            <mx:LineSeries yField="apple" displayName="Apple" name="Apple"/>
            <mx:LineSeries yField="orange" displayName="Orange" name="Orange"/>
            <mx:LineSeries yField="banana" displayName="Banana" name="Banana"/>
         </mx:series>
      </mx:LineChart>
      <mx:Legend dataProvider="{myChart}"/>
   </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="myChart" dataProvider="{myAC}" showDataTips="true">
         <mx:horizontalAxis>
            <mx:CategoryAxis categoryField="month"/>
         </mx:horizontalAxis>
         <mx:series>
            <mx:LineSeries yField="apple" displayName="Apple" name="Apple"/>
            <mx:LineSeries yField="orange" displayName="Orange" name="Orange"/>
            <mx:LineSeries yField="banana" displayName="Banana" name="Banana"/>
         </mx:series>
      </mx:LineChart>
      <mx:Legend dataProvider="{myChart}"/>
   </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}" showDataTips="true">
        <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>
```

Structure of chart data

In most cases, the data that is used as the chart's data provider is made up of scalar values. For example, objects contain a single set of fields:

```
{month: "Aug", close: 45.87, open:25.19},
```

Or XML data contains a single set of child tags in a flat structure:

```
<stock>
    <month>Aug</month>
    <close>45.87</close>
    <open>25.19</open>
</stock>
```

In these cases, you assign the data provider's fields to items in the chart by using the `xField` and `yField` for the series, or the `categoryField` for the axis; for example:

```
<mx:ColumnSeries yField="close"/>
```

```
<mx:CategoryAxis categoryField="month"/>
```

However, the structure of the chart data can be made up of more complex objects, such as objects within objects or XML with nested child tags. For example, you can embed an object within an object:

```
{month: "Aug", close: {High:45.87,Low:12.2}, open:25.19}
```

Or use nested tags in an XML object:

```
<stock>
    <date>
        <month>Aug</month>
    </date>
    <price>
        <close>45.87</close>
        <open>25.19</open>
    </price>
</stock>
```

In these cases, you cannot refer to the target data by using the flat naming such as `yField="close"`. You also cannot use dot notation. For example, `yField="values.close"` or `categoryField="data.month"` are not valid. Instead, you must use a `dataFunction` method to define which ChartItem fields are populated by the data provider.

For the CategoryAxis class, the `dataFunction` has the following signature:

```
function_name(axis:AxisBase, item:Object):Object
```

Where *`axis`* is the base class for the current axis, and *`item`* is the item in the data provider.

For the Series class, the `dataFunction` has the following signature:

```
function_name(series:Series, item:Object, fieldName:String):Object
```

Where *`series`* is a reference to the current series, *`item`* is the item in the data provider, and *`fieldName`* is the field in the current ChartItem that will be populated.

The following example creates two functions that access complex data for both the axis and the series:

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

  <mx:Script><![CDATA[
    import mx.charts.chartClasses.AxisBase;
    import mx.charts.chartClasses.Series;
    import mx.charts.CategoryAxis;
    import mx.charts.chartClasses.IAxis;
    import mx.charts.chartClasses.ChartBase;
    import mx.charts.chartClasses.CartesianTransform;

    // This data provider contains complex, nested objects.
    [Bindable]
    public var SMITH:Array = [
        {month: "Aug", close: {High:45.87,Low:12.2}, open:25.19},
```

```
        {month: "Sep", close: {High:45.74,Low:10.23}, open:35.29},
        {month: "Oct", close: {High:45.77,Low:12.13}, open:45.19},
        {month: "Nov", close: {High:46.06,Low:10.45}, open:15.59},
    ];

    private function dataFunc(series:Series, item:Object, fieldName:String):Object {
        trace("fieldName: " + fieldName);
        if(fieldName == "yValue" && series.id=="highClose")
            return(item.close.High);
        else if(fieldName == "yValue" && series.id=="lowClose")
            return(item.close.Low);
        else if(fieldName == "xValue")
            return(item.month);
        else
            return null;
      }
    private function catFunc(axis:AxisBase, item:Object):Object {
        for (var s:String in item) {
            trace(s + ":" + item[s]);
        }

        return(item.month);
    }
   ]]></mx:Script>

    <mx:ColumnChart id="chart"
        dataProvider="{SMITH}"
        showDataTips="true"
        width="100%"
        height="100%"
    >
        <mx:horizontalAxis>
            <!-- The dataFunction replaces "categoryField='month'. -->
            <mx:CategoryAxis id="h1" dataFunction="catFunc"/>
        </mx:horizontalAxis>
        <mx:series>
            <!-- The dataFunction replaces yField value, which cannot drill
            down into an object (close.High is not valid). -->
            <mx:ColumnSeries id="highClose"
                displayName="Close (High)"
                dataFunction="dataFunc"
            />

            <!-- The dataFunction replaces yField value, which cannot drill
            down into an object (close.Low is not valid). -->
            <mx:ColumnSeries id="lowClose"
                displayName="Close (Low)"
                dataFunction="dataFunc"
            />
        </mx:series>
    </mx:ColumnChart>
</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}" showDataTips="true">
            <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:Panel>

    <mx:Button id="b1"
        label="Change Data Provider"
        click="changeDataProvider()"
    />
```

```
</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);
        }

        private function removeItem():void {
            var i:int = dpac.length;
            if (i > 0) {
                dpac.removeItemAt(i - 1);
            }
        }
    ]]></mx:Script>
  <mx:Panel title="Column Chart">
     <mx:ColumnChart id="myChart"
        showDataTips="true"
        height="400"
        width="600"
        dataProvider="{dpac}"
     >
        <mx:series>
            <mx:ColumnSeries yField="A" displayName="Series 1"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
  <mx:HBox>
      <mx:Button label="Add Data Item" click="addDataItem();"/>
      <mx:Button label="Remove Data Item" click="removeItem();"/>
  </mx:HBox>
</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' `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:Panel>

<mx:Button id="b1"
    label="Toggle Series"
    click="changeData()"
/>
```

```
</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);
        }

        private function startApp():void {
            t.addEventListener(TimerEvent.TIMER, addData);
            t.start();
        }

        private function addData(e:Event):void {
            /* Add a maximum of 100 data points before user has to click
               the Start button again. */
            if (dataSet.length > 100) {
                stopApp();
            }

            dataSet.addItem( { revenue: revenue } );
            revenue += Math.random() * 10 - 5;
        }

        private function stopApp():void {
            t.stop();
            t.removeEventListener(TimerEvent.TIMER, addData);
        }
    ]]></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}"
                    displayName="Revenue"
```

```
                />
            </mx:series>
            <mx:horizontalAxis>
                <mx:LinearAxis autoAdjust="false"/>
            </mx:horizontalAxis>
        </mx:LineChart>
        <mx:Legend dataProvider="{myChart}"/>
    </mx:Panel>

    <mx:HBox>
        <mx:Button id="b1" label="Start" click="startApp()"/>
        <mx:Button id="b2" label="Stop" click="stopApp()"/>
    </mx:HBox>

</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 2: Chart Types

Adobe Flex provides different types of charting controls.

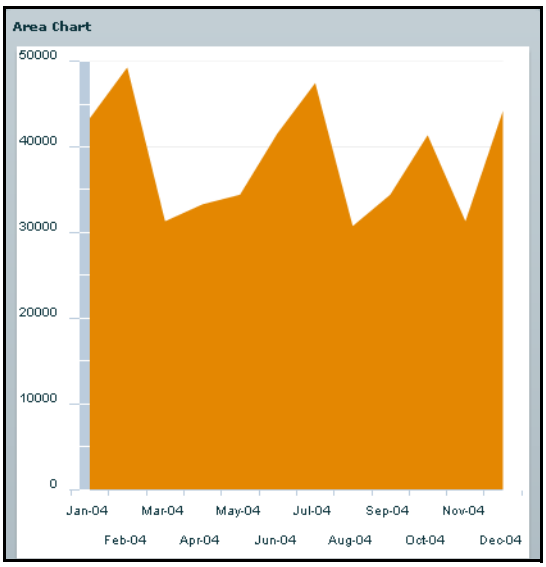
Topics

Using area charts . . . 36
Using bar charts . . . 40
Using bubble charts . . . 41
Using candlestick charts . . . 46
Using column charts . . . 50
Using HighLowOpenClose charts . . . 53
Using line charts . . . 56
Using pie charts . . . 63
Using plot charts . . . 70
Using multiple data series . . . 73
Using multiple axes . . . 75

Using area charts

You use the AreaChart control to represent data as an area bounded by a line connecting the data values. 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>
```

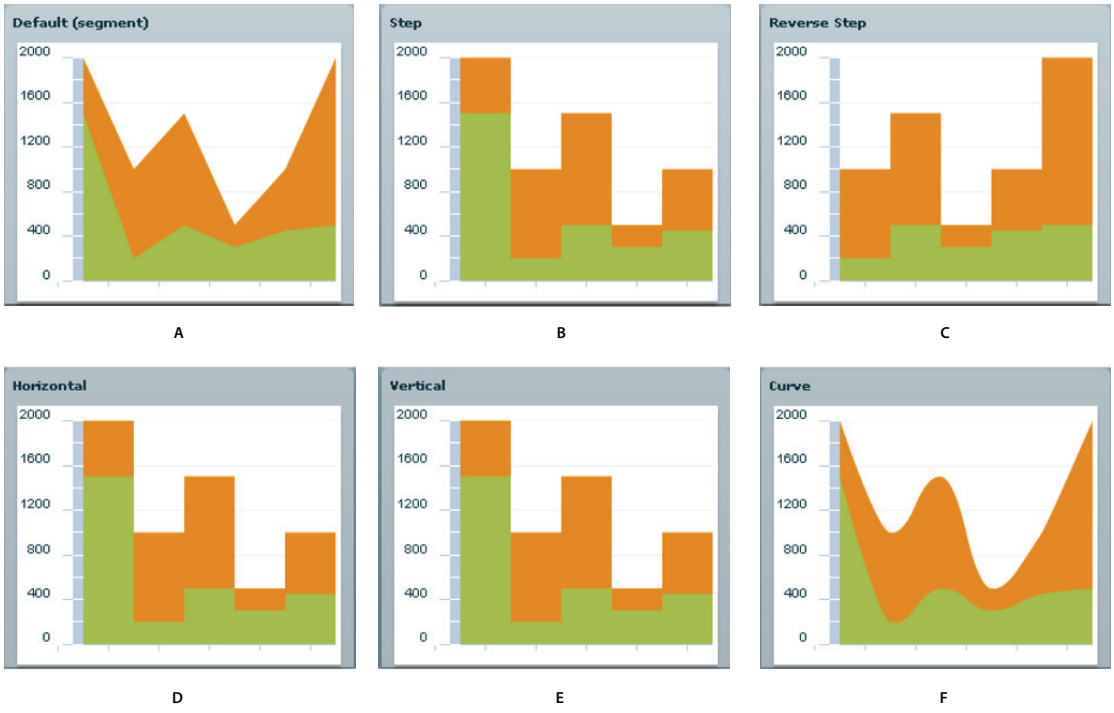
An area chart is essentially a line chart with the area underneath the line filled in; therefore, area charts and line charts share many of the same characteristics. For more information, see "Using line charts" on page 56.

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, Adobe® 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 196.

Property	Description
`form`	Specifies the way in which the data series is shown in the chart. The following values are valid: • `segment` Draws lines as connected segments that are angled to connect at each data point in the series. This is the default. • `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. • `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. • `vertical` Draws only the vertical line 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 action for each data point. • `horizontal` Draws only the horizontal line 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 action for each data point. • `curve` Draws curves between data points.

The following example shows the available forms for an AreaChart control's series:



***A.** Segment (default) **B.** Step **C.** Reverse Step **D.** Horizontal **E.** Vertical **F.** Curve*

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 198. To customize the fills of the series in an AreaChart control, you use the `areaFill` and `areaStroke` properties. For each fill, you specify a SolidColor and a Stroke object. The following example defines three custom SolidColor objects and three custom Stroke objects, and applies them to the three AreaSeries objects in the AreaChart control.

```
<?xml version="1.0"?>
<!-- charts/AreaChartFills.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 } ]);
        ]]>
    </mx:Script>

    <!-- Define custom colors for use as fills in the AreaChart control. -->
    <mx:SolidColor id="sc1" color="blue" alpha=".3"/>
    <mx:SolidColor id="sc2" color="red" alpha=".3"/>
    <mx:SolidColor id="sc3" color="green" alpha=".3"/>

    <!-- Define custom Strokes. -->
    <mx:Stroke id="s1" color="blue" weight="2"/>
    <mx:Stroke id="s2" color="red" weight="2"/>
    <mx:Stroke id="s3" color="green" weight="2"/>

    <mx:Panel title="AreaChart Control with Custom Fills Example"
        height="100%" width="100%" layout="horizontal">
        <mx:AreaChart id="Areachart"
            height="100%"
            width="70%"
            paddingLeft="5"
            paddingRight="5"
            showDataTips="true"
            dataProvider="{expensesAC}"
        >
            <mx:horizontalAxis>
                <mx:CategoryAxis categoryField="Month"/>
            </mx:horizontalAxis>

            <mx:series>
                <mx:AreaSeries
                    yField="Profit"
                    displayName="Profit"
                    areaStroke="{s1}"
                    areaFill="{sc1}"
                />
                <mx:AreaSeries
                    yField="Expenses"
                    displayName="Expenses"
                    areaStroke="{s2}"
                    areaFill="{sc2}"
                />
                <mx:AreaSeries
                    yField="Amount"
                    displayName="Amount"
                    areaStroke="{s3}"
                    areaFill="{sc3}"
                />
            </mx:series>
        </mx:AreaChart>
        <mx:Legend dataProvider="{Areachart}"/>
    </mx:Panel>
</mx:Application>
```

For more information on using fills, see "Using fills with chart controls" on page 105. For more information on using the Stroke class, see "Using strokes with chart controls" on page 99.

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 198.

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.
`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 196.

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}" 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>
```

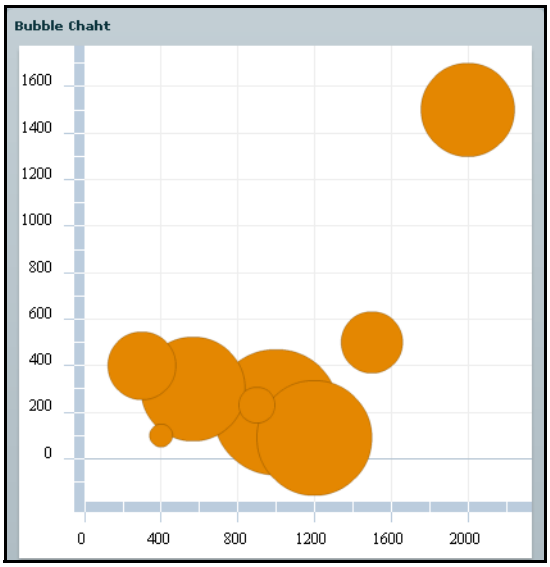
A bar chart is essentially a column chart rotated 90 degrees clockwise; therefore, bar charts and column charts share many of the same characteristics. For more information, see "Using column charts" on page 50.

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 additional properties, `minRadius` and `maxRadius`. These style properties specify the minimum and maximum radii of the chart elements, in pixels. The data point with the largest value will be less than the `maxRadius` and the data point with the smallest value will be larger than the `minRadius` property. For example, if you set the minRadius to 10 and the maxRadius to 50, all data points will have a radius between 10 and 50. The default value for `maxRadius` is 50 pixels. The default value for `minRadius` is 0 pixels. You can also control the `minRadius` and `maxRadius` properties by using properties of the same name on the BubbleSeries class.

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"
        minRadius="1"
        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>
```

To customize the styles of the bubbles in a BubbleChart control, you use the `fill` and `stroke` properties. You pass these a SolidColor and a Stroke object, respectively. The following example defines a custom SolidColor object and a custom Stroke object, and applies them to the BubbleSeries object in the BubbleChart control.

```
<?xml version="1.0"?>
<!-- charts/BubbleFills.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 }]);
    ]]>
    </mx:Script>

    <!-- Define custom color and line style for the bubbles. -->
    <mx:SolidColor id="sc1" color="blue" alpha=".3"/>
    <mx:Stroke id="stroke1" color="blue" weight="2"/>

    <mx:Panel title="BubbleChart Control with Custom Bubble Styles">
        <mx:BubbleChart id="myChart" showDataTips="true">
            <mx:series>
                <mx:BubbleSeries
                    dataProvider="{s1}"
                    displayName="series1"
                    xField="x"
                    yField="y"
```

```
                    radiusField="r"
                    fill="{sc1}"
                    stroke="{stroke1}"
                />
            </mx:series>
        </mx:BubbleChart>
        <mx:Legend dataProvider="{myChart}"/>
    </mx:Panel>
</mx:Application>
```

For more information on using fills, see "Using fills with chart controls" on page 105. For more information on using the Stroke class, see "Using strokes with chart controls" on page 99.

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.

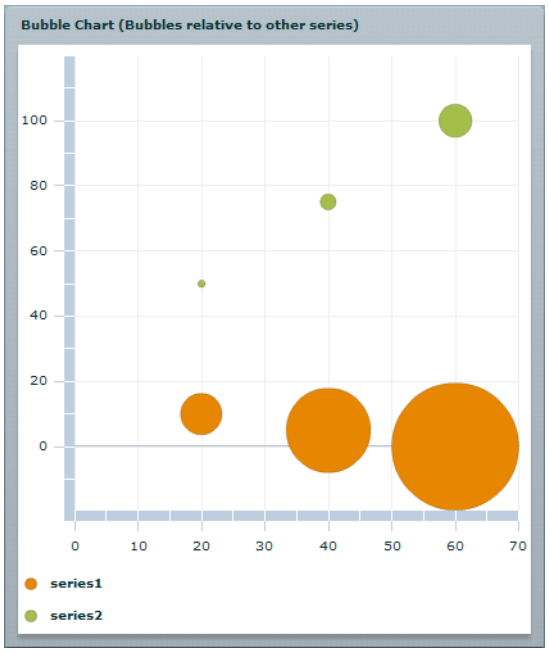
For example, you have two series, A and B. Series A has bubbles with radius values of 10, 20, and 30. Series B has bubbles with radius values of 2, 4, and 8. Your BubbleChart control can display bubbles in series A that are all larger than the bubbles in series B. You can also design a BubbleChart control 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 2 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>

    <mx:Panel title="Bubble Chart (Bubbles relative to other series)">
        <mx:BubbleChart id="myChart"
            showDataTips="true"
        >
            <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:

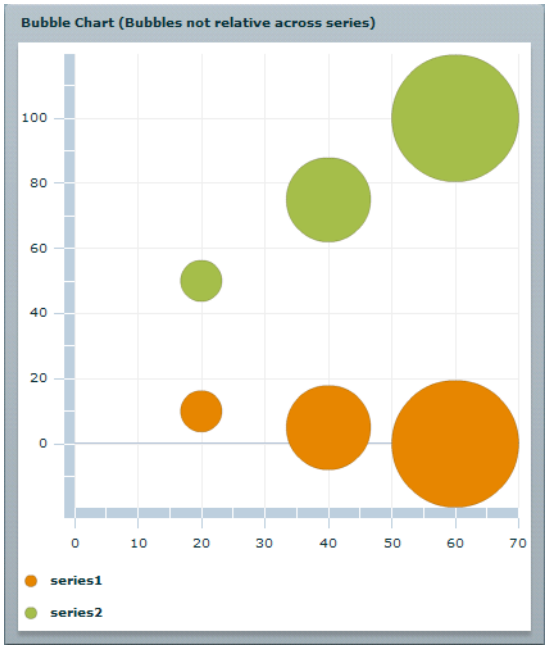


If you want the size of the bubbles to be relative to each other in their own series, but not to other series, use a different radial axis for each series. To do this, you add a `<mx:radialAxis>` child tag to the `<mx:BubbleSeries>` tag in the MXML. This creates two series in the BubbleChart control whose bubble sizes are independent of one another. The following example shows a BubbleChart control with two independant series:

```
<?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":1 },
                {"x": 40, "y": 75, "r":2 },
                {"x": 60, "y": 100, "r":3 } ]);

    ]]>
    </mx:Script>

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

The following example shows a BubbleChart control with two series whose bubble sizes are independant of one another:



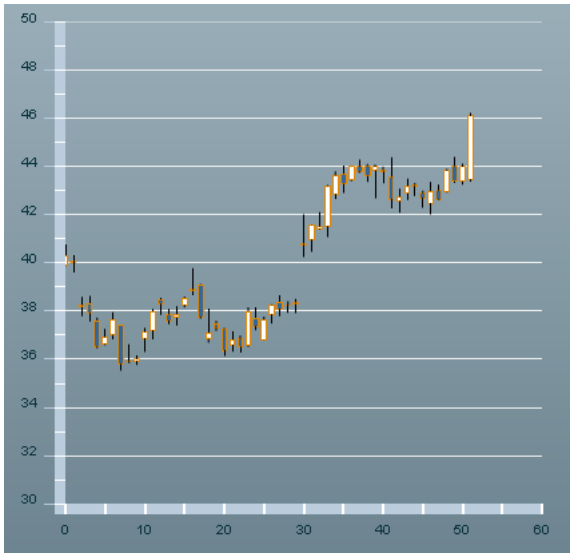
Using candlestick charts

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 represent 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 53.

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

The following shows an example of a CandlestickChart control:

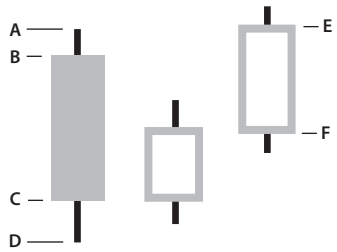


The following table describes the properties of the CandlestickSeries chart series that you commonly use to define your chart:

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.
`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 the properties that define the appearance of the candle. 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:



***A.** highField **B.** openField **C.** closeField **D.** lowField **E.** closeField **F.** openField*

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:

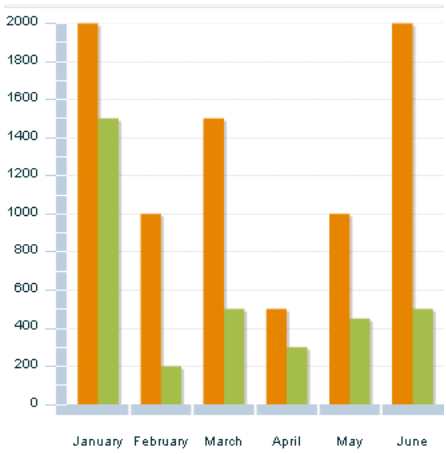
```
<?xml version="1.0"?>
<!-- charts/CandlestickStyles.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [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 198.

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 196.

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}" showDataTips="true">
       <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 customize the styles of the columns in a ColumnChart control, you specify a SolidColor and a Stroke object for the `fill` and `stroke` properties, respectively. The following example defines a custom SolidColor object and a custom Stroke object, and applies them to the ColumnSeries object in the ColumnChart control.

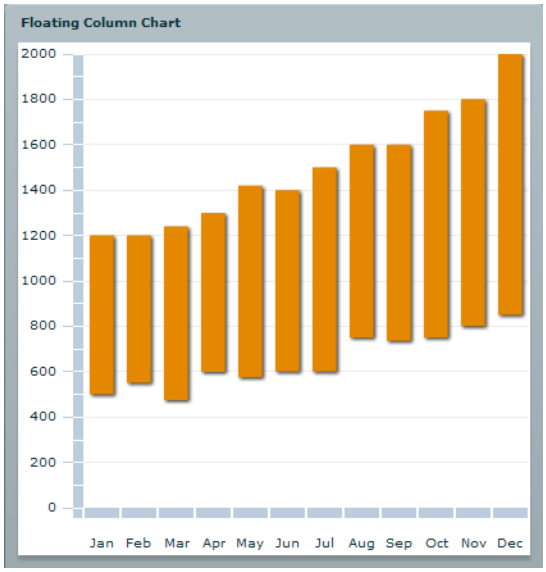
```
<?xml version="1.0"?>
<!-- charts/BasicColumnFills.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script><![CDATA[
        import mx.collections.ArrayCollection;
        [Bindable]
        public var expenses:ArrayCollection = new ArrayCollection([
            {Month:"Jan",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

    <!-- Define custom colors for use as column fills. -->
    <mx:SolidColor id="sc1" color="blue" alpha=".3"/>
    <mx:SolidColor id="sc2" color="red" alpha=".3"/>

    <!-- Define custom Strokes for the columns. -->
    <mx:Stroke id="s1" color="blue" weight="2"/>
    <mx:Stroke id="s2" color="red" weight="2"/>

    <mx:Panel title="ColumnChart Control with Custom Column Styles">
        <mx:ColumnChart id="myChart" dataProvider="{expenses}" showDataTips="true">
            <mx:horizontalAxis>
                <mx:CategoryAxis
                    dataProvider="{expenses}"
                    categoryField="Month"
                />
            </mx:horizontalAxis>
            <mx:series>
                <mx:ColumnSeries
                    xField="Month"
                    yField="Profit"
                    displayName="Profit"
```

```
                fill="{sc1}"
                stroke="{s1}"
            />
            <mx:ColumnSeries
                xField="Month"
                yField="Expenses"
                displayName="Expenses"
                fill="{sc2}"
                stroke="{s2}"
            />
        </mx:series>
    </mx:ColumnChart>
    <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

For more information on using fills, see "Using fills with chart controls" on page 105. For more information on using the Stroke class, see "Using strokes with chart controls" on page 99.

Creating floating column charts

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>
```

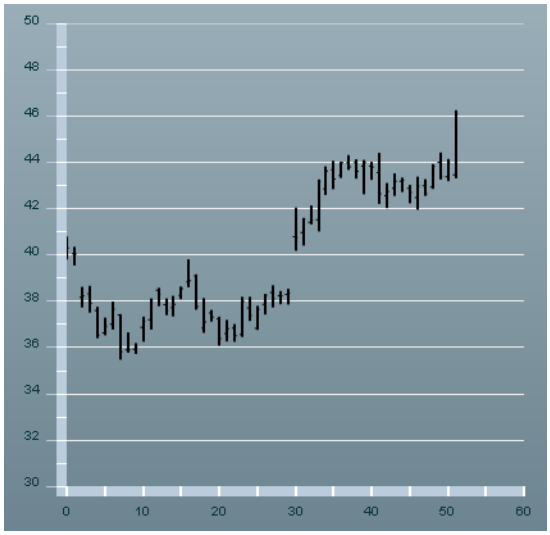
For more information, see "Using the minField property" on page 196.

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 "Using candlestick charts" on page 46.

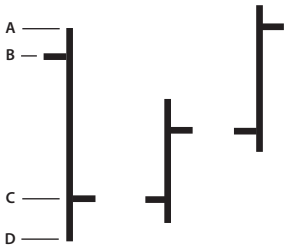
You use the HLOCSeries with the HLOCChart control to define the data for HighLowOpenClose charts. The following example shows an HLOCChart control:



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.
`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:



***A.** highField **B.** openField **C.** closeField **D.** lowField*

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:

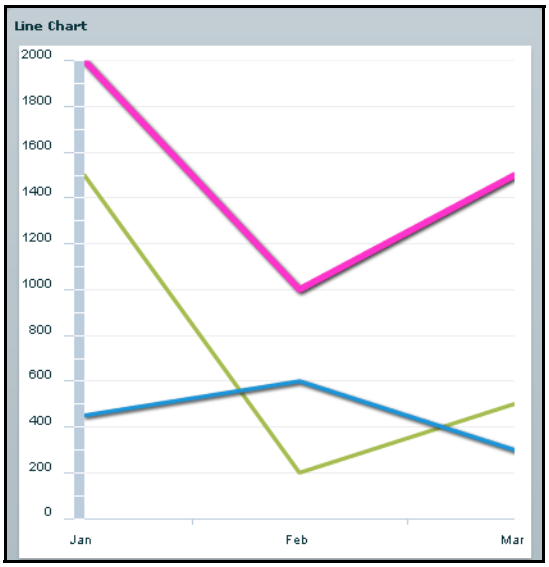
```
<?xml version="1.0"?>
<!-- charts/HLOCStyled.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;

     [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:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</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.

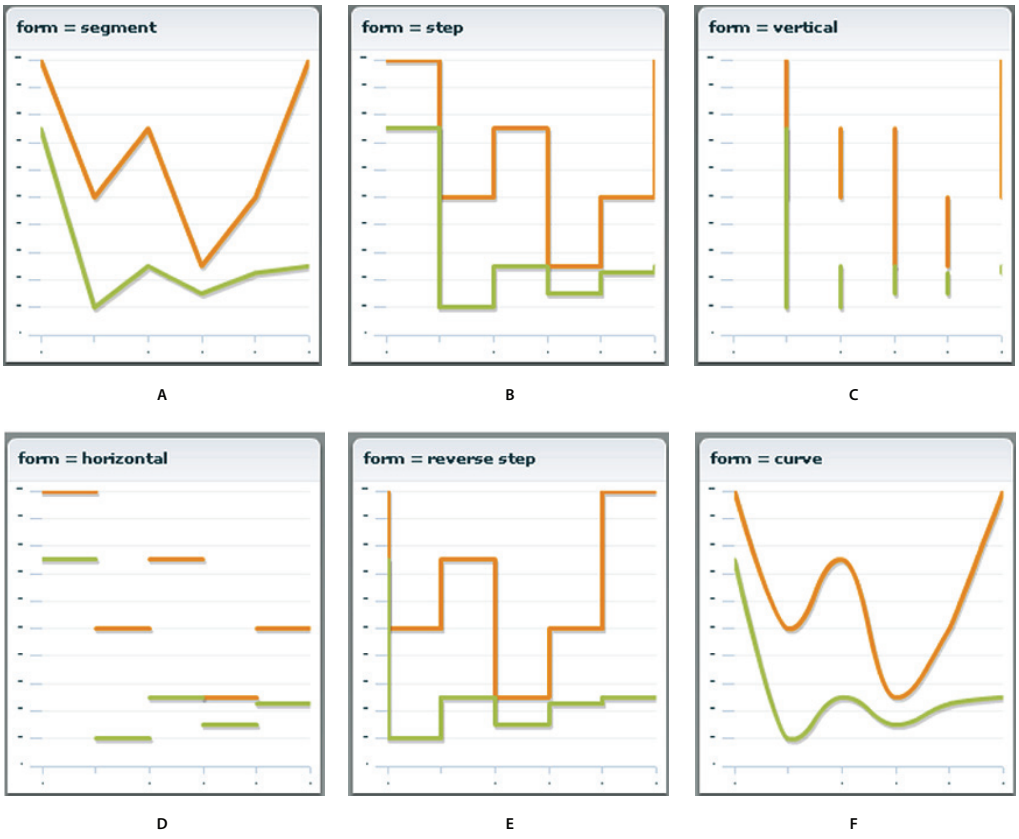
The following example shows a simple LineChart control with three series:



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 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 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: • `segment` Draws lines as connected segments that are angled to connect at each data point in the series. This is the default. • `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. • `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. • `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. • `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. • `curve` Draws curves between data points.

The following example shows the available forms for a LineChart control's series:



***A.** segment (default) **B.** step **C.** vertical **D.** horizontal **E.** reverseStep **F.** 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>
```

You can also modify lines in a LineChart with ActionScript. You define a new Stroke, and apply it to the LineSeries by setting the `lineStroke` style property with the `setStyle()` method. For example:

```
var s1:Stroke = new Stroke(0x0099FF,20,.2); //First 3 arguments are color, weight, and alpha.
series1.setStyle("lineStroke", s1);
```

For more information on using the Stroke class in charts, see "Using strokes with chart controls" on page 99.

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

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

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

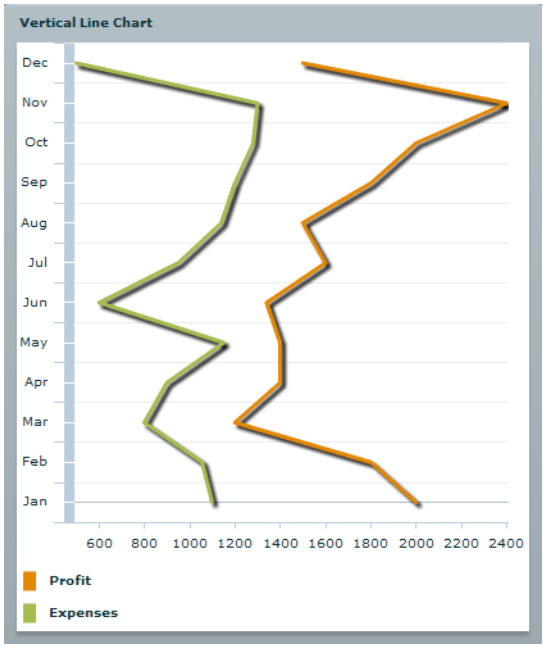
You can also specify a programmatic renderer (or *skin*) class for each series by setting the `lineSegmentRenderer` property of the LineSeries. The default renderer is the LineRenderer, but Flex also applies a shadow filter on all series. If you remove the shadow filter, as the previous example shows, 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}" showDataTips="true">
       <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:Legend dataProvider="{myChart}"/>
  </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 132.

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 arranged correctly in the data provider. If it is not arranged correctly, Flex renders a zig-zagging line up and down the chart as it connects those dots according to position in the data provider.

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

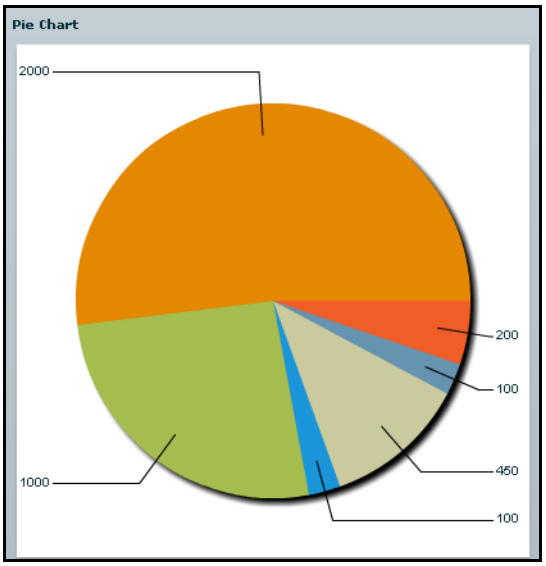
```
<?xml version="1.0"?>
<!-- charts/VerticalLineChart.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, 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 PieChart control with callouts:



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 DataTip objects 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>
```

To customize the colors of the wedges in a PieChart control, you use the `fills` property. You pass this an Array of SolidColor objects. The following example defines an Array of custom SolidColor objects, and applies it to the PieSeries object in the PieChart control.

```
<?xml version="1.0"?>
<!-- charts/PieFilling.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>

    <!-- Define custom colors for use as pie wedge fills. -->
    <mx:SolidColor id="sc1" color="blue" alpha=".3"/>
    <mx:SolidColor id="sc2" color="red" alpha=".3"/>
    <mx:SolidColor id="sc3" color="green" alpha=".3"/>
    <mx:SolidColor id="sc4" color="gray" alpha=".3"/>
    <mx:SolidColor id="sc5" color="black" alpha=".3"/>
    <mx:SolidColor id="sc6" color="yellow" alpha=".3"/>

    <mx:Panel title="PieChart control with custom fills">
        <mx:PieChart id="pie"
            dataProvider="{expenses}"
            showDataTips="true"
        >
            <mx:series>
                <mx:PieSeries
                    field="Amount"
                    nameField="Expense"
                    labelPosition="callout"
                    fills="{[sc1, sc2, sc3, sc4, sc5, sc6]}"
                />
            </mx:series>
        </mx:PieChart>
        <mx:Legend dataProvider="{pie}"/>
    </mx:Panel>
</mx:Application>
```

For more information on using fills, see “Using fills with chart controls” on page 105.

You can also customize the lines that outline each wedge by using the `stroke` property of a PieSeries object. You set this property to an instance of the Stroke class. For more information on using the Stroke class, see “Using strokes with chart controls” on page 99.

Using data labels with PieChart controls

PieChart controls support data labels that display information about each data point. All charts support DataTip objects, 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 DataTip objects 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. 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. You commonly set the value of the `labelPosition` property to `insideWithCallout` when the actual size of your chart is flexible and users might resize it. 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.

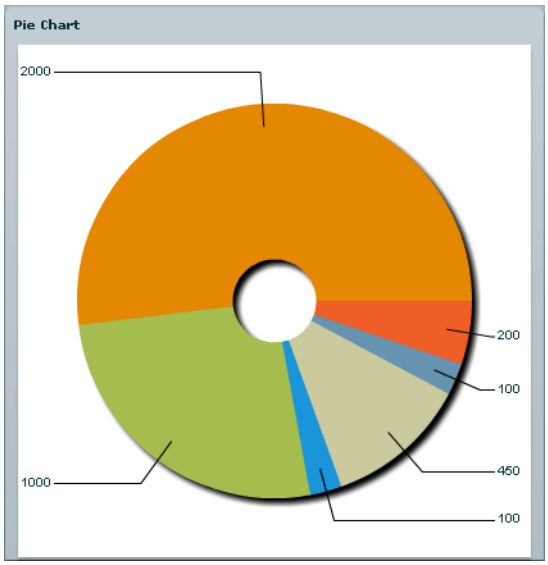
The following table describes the properties of the PieSeries object that you can use to manipulate the appearance of labels:

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 with chart controls" on page 99.
`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 label values" on page 176.

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:

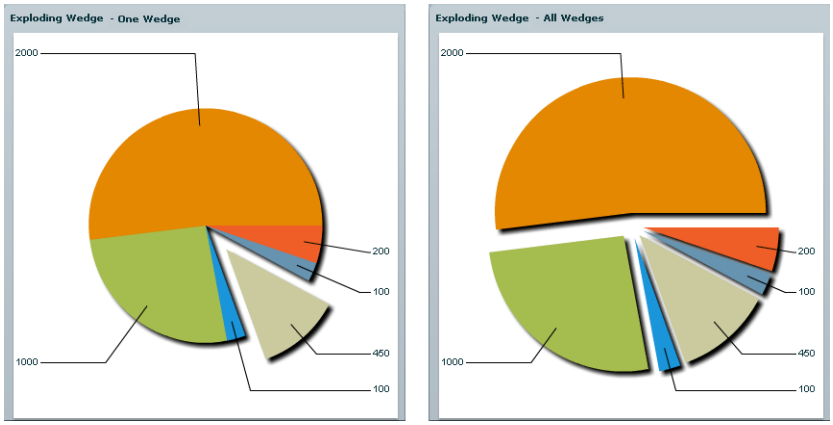
```
<?xml version="1.0"?>
<!-- charts/DoughnutPie.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Expense:"Taxes", Amount:2000},
        {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>
```

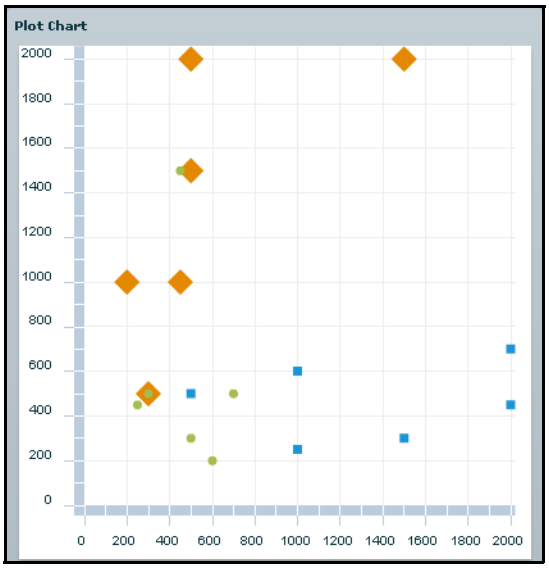
To explode one or more wedges of the pie, you use an Array of `explodeRadius` values. Each value in the Array applies to the corresponding data point. In the following example, the fourth data point, the Car expense, is exploded:

```
<?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 PlotChart control with three series:



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>
```

To customize the styles of the points in a PlotChart control, you define a SolidColor and a Stroke object for the `fill` and `stroke` properties, respectively. The following example defines three SolidColor objects and three custom Stroke objects, and applies them to the PlotSeries objects in the PlotChart control.

```
<?xml version="1.0"?>
<!-- charts/PlotFills.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>

    <!-- Define custom colors for use as plot point fills. -->
    <mx:SolidColor id="sc1" color="blue" alpha=".3"/>
    <mx:SolidColor id="sc2" color="red" alpha=".3"/>
    <mx:SolidColor id="sc3" color="green" alpha=".3"/>

    <!-- Define custom Strokes. -->
    <mx:Stroke id="s1" color="blue" weight="1"/>
    <mx:Stroke id="s2" color="red" weight="1"/>
    <mx:Stroke id="s3" color="green" weight="1"/>

    <mx:Panel title="Plot Chart with custom fills">
        <mx:PlotChart id="myChart"
            dataProvider="{expenses}"
            showDataTips="true"
        >
            <mx:series>
```

```
                <mx:PlotSeries
                    xField="Expenses"
                    yField="Profit"
                    displayName="Plot 1"
                    fill="{sc1}"
                    stroke="{s1}"
                />
                <mx:PlotSeries
                    xField="Amount"
                    yField="Expenses"
                    displayName="Plot 2"
                    fill="{sc2}"
                    stroke="{s2}"
                />
                <mx:PlotSeries
                    xField="Profit"
                    yField="Amount"
                    displayName="Plot 3"
                    fill="{sc3}"
                    stroke="{s3}"
                />
            </mx:series>
        </mx:PlotChart>
        <mx:Legend dataProvider="{myChart}"/>
    </mx:Panel>
</mx:Application>
```

For more information on using fills, see "Using fills with chart controls" on page 105. For more information on using the Stroke class, see "Using strokes with chart controls" on page 99.

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},
```

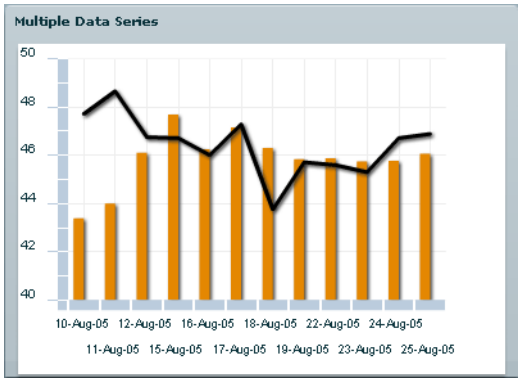
```
        {Month:"April", Profit:500, Expenses:300, Amount:500},
        {Month:"May", Profit:1000, Expenses:450, Amount:250},
        {Month:"June", Profit:2000, Expenses:500, Amount:700}
    ]);
   ]]>
  </mx:Script>

  <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 135.

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 (using a ColumnSeries) and the other represented by a line (using a LineSeries):



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

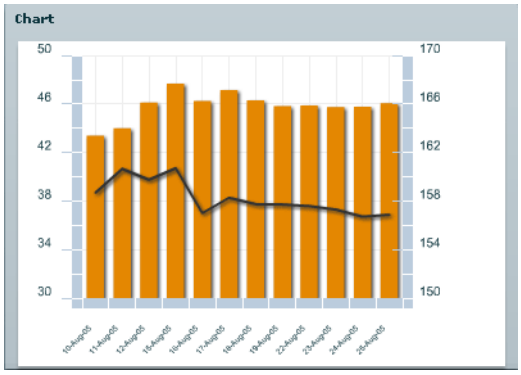
Using multiple series in the same chart works best when the data points are in a similar range (such as a stock price and its moving average). When the data points are in numerically very different ranges, the chart can be difficult to understand because the data is shown on a single axis. The solution 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 75.

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 can 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 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, as follows:

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

And then bind it to the series, like this:

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

Even if the `verticalAxisRenderers` or `horizontalAxisRenderers` properties 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 166.

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 3: Formatting Charts

You can customize the look and feel of your Adobe® Flex® charting controls. 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 627 in *Adobe Flex 3 Developer Guide*.

Topics

Applying chart styles. 81
Setting padding properties. 91
Formatting tick marks 96
Formatting axis lines. 97
Using strokes with chart controls 99
Using fills with chart controls 105
Using filters with chart controls 116
Using chart grid lines 121
Positioning chart axes. 127
Rotating chart axis labels 129
Skinning ChartItem objects. 132
Formatting Legend controls 140

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 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.

To determine if an object's properties can be styled by using CSS, check to see if the class or its parent implements the IStyleClient interface. If it does, then you can set the values of the object's style properties with CSS. If it does not, then the class does not participate in the styles subsystem and you therefore cannot set its properties with CSS. In that case, the properties are public properties of the class and not style properties. You can apply properties with CSS by using pre-defined styles for some classes, such as the `verticalAxisStyleName` and `horizontalAxisStyleName`. For more information, see "Using predefined axis style properties" on page 84.

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 title="Setting Styles on Series">
     <mx:ColumnChart id="column" dataProvider="{expenses}" showDataTips="true">
        <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 the values in this Array to the pie wedges. 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 title="Setting Pie Wedge Fills with CSS">
     <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, the following two class selectors 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 title="Using Predefined Axis Styles">
     <mx:ColumnChart id="column" dataProvider="{expenses}" showDataTips="true">
        <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. Note that any time you want to use an AxisRenderer, you must explicitly set the axis to which it is applied with the renderer's `axis` property.

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

  <mx:Panel title="AxisRenderer Class Selectors example">
     <mx:ColumnChart id="column" dataProvider="{expenses}" showDataTips="true">

        <mx:horizontalAxisRenderers>
           <mx:AxisRenderer
                axis="{a1}"
                styleName="myStyle"
           />
        </mx:horizontalAxisRenderers>

        <mx:horizontalAxis>
           <mx:CategoryAxis
                id="a1"
                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 589 in *Adobe Flex 3 Developer Guide*.

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

Notice that any time you want to use an AxisRenderer, you must explicitly set the axis to which it is applied with the renderer's `axis` property.

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 145.

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
            axis="{a1}"
            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 id="a1"
            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/butterfly.gif")]
public var butterfly: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 butterfly();
    b.alpha = .2;
    b.height = 150;
    b.width = 150;

    myChart.backgroundElements = [ b ];
}
]]></mx:Script>
<mx:Panel title="Annotation Elements Example">
    <mx:ColumnChart id="myChart" dataProvider="{expenses}" showDataTips="true">
        <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/butterfly.gif')"
                alpha=".2"
           />
        </mx:backgroundElements>

     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
  <mx:Button id="b1"
    click="updateGridLines()"
    label="Update Grid Lines"
  />
</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="addButterfly()">
  <mx:Script><![CDATA[
      import mx.collections.ArrayCollection;
      import mx.graphics.SolidColor;
      import mx.charts.GridLines;
      import mx.charts.ColumnChart;

      [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/butterfly.gif")]
      public var butterfly:Class;

      public function addButterfly():void {
         var b:Object = new butterfly();
         b.alpha = .2;
         myChart.backgroundElements = [ b ];
      }
  ]]></mx:Script>

  <mx:Panel title="Background Elements with ActionScript">
     <mx:ColumnChart id="myChart" dataProvider="{expenses}" showDataTips="true">
        <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 chart grid lines" on page 121.

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 264.

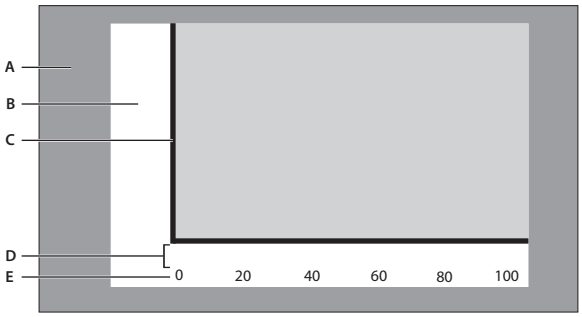
Setting padding properties

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 control 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:



***A.** Padding area **B.** Gutter **C.** Axis **D.** Label gap **E.** Axis label*

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: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>
     ColumnChart {
        gutterLeft:50;
        gutterRight:50;
        gutterBottom:50;
        paddingTop:20;
     }
  </mx:Style>

  <mx:Panel title="Setting Gutter Properties as Styles">
     <mx:ColumnChart id="column" dataProvider="{expenses}" showDataTips="true">
        <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 title="Setting Gutter Properties">
        <mx:ColumnChart id="myChart"
            dataProvider="{expenses}"
            gutterLeft="50"
            gutterRight="50"
            gutterBottom="50"
            gutterTop="20"
            showDataTips="true"
        >
            <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 title="Setting Label Gap Properties">
        <mx:ColumnChart id="myChart"
            dataProvider="{expenses}"
            gutterLeft="50"
            gutterRight="50"
            gutterBottom="50"
```

```
            gutterTop="20"
            showDataTips="true"
        >
            <mx:horizontalAxis>
                <mx:CategoryAxis
                    id="a1"
                    dataProvider="{expenses}"
                    categoryField="Month"
                />
            </mx:horizontalAxis>

            <mx:horizontalAxisRenderers>
                <mx:AxisRenderer
                    axis="{a1}"
                    labelGap="20"
                />
            </mx:horizontalAxisRenderers>

            <mx:verticalAxisRenderers>
                <mx:AxisRenderer
                    axis="{a1}"
                    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>
```

You can specify the alignment of axis labels by using the `labelAlign` property on the AxisRenderer. For horizontal axis labels, you can specify `center`, `left`, and `right`. For example, in a ColumnChart control, if you set the value of the `labelAlign` property to `left`, Flex renders the labels at the `left` of the bottoms of the columns. For vertical axis labels, you can specify `center`, `top`, and `bottom`.

How different the locations of the labels are depends on the available space (for example, the width of the columns) and the font size of the labels.

The following example lets you change the label alignment for the vertical and horizontal axes.

```
<?xml version="1.0"?>
<!-- charts/ToggleLabelAlignment.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 aligns:Array = ["center", "left", "right"];
    [Bindable]
    private var aligns2:Array = ["center", "top", "bottom"];

    private function toggleLabelAlignment(s:String):void {
        if ( s == 'hor' ) {
            myHAR.setStyle("labelAlign", cb1.selectedItem);
        } else if ( s == 'ver' ) {
            myVAR.setStyle("labelAlign", cb2.selectedItem);
        }
    }
]]></mx:Script>

<mx:Panel title="Toggling Label Alignment">
   <mx:ColumnChart id="myChart" dataProvider="{expenses}" showDataTips="true">
      <mx:horizontalAxis>
         <mx:CategoryAxis id="haxis" categoryField="Month"/>
      </mx:horizontalAxis>

      <mx:verticalAxis>
          <mx:LinearAxis id="vaxis"/>
      </mx:verticalAxis>

      <mx:horizontalAxisRenderers>
          <mx:AxisRenderer id="myHAR" axis="{haxis}"/>
      </mx:horizontalAxisRenderers>

      <mx:verticalAxisRenderers>
          <mx:AxisRenderer id="myVAR" axis="{vaxis}"/>
      </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:Form>
      <mx:FormItem label="Horizontal Labels">
          <mx:ComboBox id="cb1"
              dataProvider="{aligns}"
              change="toggleLabelAlignment('hor')"
          />
      </mx:FormItem>
      <mx:FormItem label="Vertical Labels">
          <mx:ComboBox id="cb2"
              dataProvider="{aligns2}"
              change="toggleLabelAlignment('ver')"
          />
      </mx:FormItem>
   </mx:Form>
   <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 to 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 FRED: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="{FRED}"
        showDataTips="true"
    >
    <mx:horizontalAxisRenderers>
        <mx:AxisRenderer axis="{a2}" styleName="myAxisStyle"/>
    </mx:horizontalAxisRenderers>

    <mx:verticalAxis>
        <mx:LinearAxis id="a1" minimum="30" maximum="50"/>
    </mx:verticalAxis>

    <mx:horizontalAxis>
        <mx:LinearAxis id="a2"/>
    </mx:horizontalAxis>

    <mx:series>
        <mx:HLOCSeries
            dataProvider="{FRED}"
            openField="open"
            highField="high"
            lowField="low"
            closeField="close"
            displayName="Ticker Symbol: FRED"
        />
     </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 id="myChart" dataProvider="{expenses}" showDataTips="true">
            <mx:horizontalAxis>
                <mx:CategoryAxis
```

```
                    id="a1"
                    dataProvider="{expenses}"
                    categoryField="Month"
                />
            </mx:horizontalAxis>
            <mx:horizontalAxisRenderers>
                <mx:AxisRenderer
                    axis="{a1}"
                    showLine="false"/>
            </mx:horizontalAxisRenderers>
            <mx:verticalAxisRenderers>
                <mx:AxisRenderer
                    axis="{a1}"
                    showLine="false"
                />
            </mx:verticalAxisRenderers>
            <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>
```

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 id="myChart" dataProvider="{expenses}" showDataTips="true">
            <mx:horizontalAxis>
                <mx:CategoryAxis
```

```
                id="a1"
                dataProvider="{expenses}"
                categoryField="Month"
            />
        </mx:horizontalAxis>

        <mx:verticalAxis>
            <mx:LinearAxis id="a2"/>
        </mx:verticalAxis>

        <mx:horizontalAxisRenderers>
            <mx:AxisRenderer axis="{a1}">
                <mx:axisStroke>{axisStroke}</mx:axisStroke>
            </mx:AxisRenderer>
        </mx:horizontalAxisRenderers>

        <mx:verticalAxisRenderers>
            <mx:AxisRenderer axis="{a2}">
                <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:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

For more information about strokes, see "Using strokes with chart controls" on page 99.

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 chart axes" on page 127.

You can also apply filters to axis lines to further customize their appearance. For more information, see "Using filters with chart controls" on page 116.

Using strokes with chart controls

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}" showDataTips="true">
        <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"
                axis="{a1}"
            >
                <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
                id="a1"
                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>
```

Notice that any time you use an AxisRenderer, you must explicitly set the axis to which it is applied with the renderer's `axis` property.

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 87.

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}" showDataTips="true">
        <mx:horizontalAxis>
```

```
            <mx:CategoryAxis
                  id="a1"
                  dataProvider="{expenses}"
                  categoryField="Month"
            />
         </mx:horizontalAxis>

         <mx:horizontalAxisRenderers>
            <mx:AxisRenderer id="har1" axis="{a1}">
                  <mx:axisStroke>{baseAxisStroke}</mx:axisStroke>
            </mx:AxisRenderer>
         </mx:horizontalAxisRenderers>

         <mx:verticalAxisRenderers>
            <mx:AxisRenderer id="var1" axis="{a1}">
                  <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 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 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 190.

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 205.

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}" showDataTips="true">
        <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}" showDataTips="true">
```

```
        <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:Legend dataProvider="{myChart}"/>
  </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 205.

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 shows how you 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 id="myChart" 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" displayName="Amount"/>
        </mx:series>
     </mx:BarChart>
  </mx:Panel>
  <mx:Legend dataProvider="{myChart}"/>
</mx:Application>
```

Setting fills with CSS

You can use the `fill` or `fills` style properties in an `<mx:Style>` declaration by using CSS syntax. You can use a type or a 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 id="myChart"
       dataProvider="{expenses}"
       showDataTips="true"
       styleName="myBarChartStyle"
    >
       <mx:verticalAxis>
          <mx:CategoryAxis
               dataProvider="{expenses}"
               categoryField="Expense"
          />
       </mx:verticalAxis>
       <mx:series>
          <mx:BarSeries xField="Amount" displayName="Amount"/>
       </mx:series>
    </mx:BarChart>
  </mx:Panel>
  <mx:Legend dataProvider="{myChart}"/>
</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}" showDataTips="true">
        <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:Legend dataProvider="{myChart}"/>
  </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}" showDataTips="true">
        <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:Legend dataProvider="{myChart}"/>
  </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: • `color` Specifies a color value. • `alpha` Specifies the transparency. Valid values are 0 (invisible) through 1 (opaque). The default value is 1. • `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 id="myChart"
       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" displayName="Amount"/>
       </mx:series>
    </mx:BarChart>
    <mx:Legend dataProvider="{myChart}"/>
  </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 id="myChart" 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" displayName="Amount"/>
        </mx:series>
     </mx:BarChart>
     <mx:Legend dataProvider="{myChart}"/>
  </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>
```

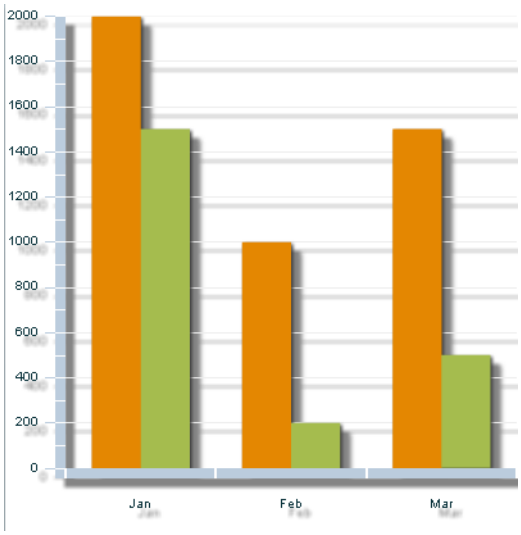
In this example, you make the last gradient color in the series partially transparent by setting its `alpha` property to .2.

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 title="ColumnChart with drop shadow example">
     <mx:ColumnChart id="column" dataProvider="{expenses}" showDataTips="true">
        <!-- Add a custom drop shadow filter to the ColumnChart control. -->
        <mx:filters>
           <mx: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>
```

For more information on using filters, see "Using filters in Flex" on page 642 in *Adobe Flex 3 Developer Guide*.

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 default drop shadow filter on the PieSeries.

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}" showDataTips="true">
     <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>
```

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 title="Applying Multiple Filters">
    <mx:ColumnChart id="myChart" dataProvider="{expenses}" showDataTips="true">
        <mx:backgroundElements>
            <mx:GridLines id="myGridlines"
                horizontalChangeCount="1"
                verticalChangeCount="1"
                direction="both"
            />
        </mx:backgroundElements>
        <mx:horizontalAxis>
            <mx:CategoryAxis
                id="a1"
                dataProvider="{expenses}"
                categoryField="Month"
            />
        </mx:horizontalAxis>

        <mx:horizontalAxisRenderers>
            <mx:AxisRenderer
                id="myAxisRenderer"
                placement="bottom"
                canDropLabels="true"
                axis="{a1}"
            >
                <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 chart 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"
        showDataTips="true"
```

```
        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}" showDataTips="true">
        <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 with chart controls" on page 99. For more information on using the `backgroundElements` and `annotationElements` properties, see "Using ChartElement objects" on page 88.

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 chart objects.

Formatting chart 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"
        showDataTips="true"
        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:Panel title="BubbleChart control with no grid lines">
    <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:Legend dataProvider="{bc}"/>
  </mx:Panel>
</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 116.

Formatting chart 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"
        showDataTips="true"
        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 chart 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"
        showDataTips="true"
        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 chart 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 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"?>
```

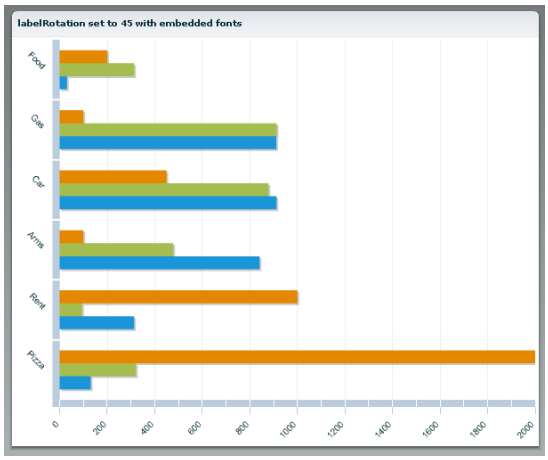
```
<!-- charts/AxisPlacementExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, Expenses:1500},
        {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}" showDataTips="true">
        <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 chart axis labels

You can rotate axis labels by 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 title="Rotated Axis Labels">
     <mx:ColumnChart id="column" dataProvider="{expenses}" showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
                title="FY 2006"
                id="a1"
           />
        </mx:horizontalAxis>

        <mx:horizontalAxisRenderers>
           <mx:AxisRenderer labelRotation="45" axis="{a1}"/>
        </mx:horizontalAxisRenderers>
        <mx:verticalAxisRenderers>
           <mx:AxisRenderer labelRotation="45" axis="{a1}"/>
        </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>
```

To change the appearance of the axis labels, you can also use the `labelRenderer` property of the AxisRenderer class. This lets you specify a class that defines the appearance of the label. The class must extend UIComponent and implement the IDataRenderer and IFlexDisplayObject interfaces. Its `data` property will be the label used in the chart. Typically, you write an ActionScript class that extends the ChartLabel class for the label renderer.

The following example defines a custom component inline by using the `<mx:Component>` tag. It adds ToolTip objects to the labels along the chart's horizontal axis so that the values of the longer labels are not truncated.

```
<?xml version="1.0"?>
<!-- charts/LabelRendererWithToolTips.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" initialize="setupDP()"
width="550" height="600">
    <mx:Script>
     <![CDATA[
        import mx.collections.ArrayCollection;
        import mx.charts.ColumnChart;
```

```
    [Bindable]
    private var ac:ArrayCollection;

    public function setupDP():void{
        ac =  new ArrayCollection([
          [ "Label 1 is short.", 200000],
          [ "Label 2 is a fairly long label.", 150000],
          [ "Label 3 is an extremely long label. It contains 95 characters " +
            "and will likely be truncated.", 40000]
        ]);
    }
 ]]>
</mx:Script>

<mx:VBox>
    <mx:ColumnChart id="bc1"
        showDataTips="true"
        dataProvider="{ac}"
    >
        <mx:series>
            <mx:ColumnSeries xField="0" yField="1"/>
        </mx:series>
        <mx:verticalAxis>
            <mx:LinearAxis id="va1"/>
        </mx:verticalAxis>
        <mx:horizontalAxis >
            <mx:CategoryAxis id="ha1"
                dataProvider="{ac}"
                categoryField="0"
            />
        </mx:horizontalAxis>
        <mx:horizontalAxisRenderers>
            <mx:AxisRenderer
                axis="{ha1}"
                canDropLabels="false"
            >
                <mx:labelRenderer>
                    <mx:Component>
                        <mx:Label toolTip="{this.myTip}">
                            <mx:Script><![CDATA[
                                [Bindable]
                                private var myTip:String;

                                override public function set data(value:Object):void{
                                    if(value == null)
                                        return;
                                    myTip = value.text;
                                    var length:int = value.text.toString().length;
                                    if (length > 20) {
                                    text = value.text.toString().substr(0, 20) + "...";
                                    } else {
                                        text = value.text;
                                    }
                                }
                            ]]></mx:Script>
                        </mx:Label>
                    </mx:Component>
                </mx:labelRenderer>
            </mx:AxisRenderer>
        </mx:horizontalAxisRenderers>
        <mx:verticalAxisRenderers>
```

```
                <mx:AxisRenderer
                    axis="{va1}"
                    canDropLabels="false"
                />
            </mx:verticalAxisRenderers>
        </mx:ColumnChart>
       <mx:Label id="l1" color="white" text="Hover over the horizontal axis's labels to see
the entire title rendered as a ToolTip."/>
    </mx:VBox>
</mx:Application>
```

For an example of an ActionScript class that extends ChartLabel, see "Adding axis titles" on page 159.

Skinning ChartItem objects

A ChartItem object represents a data point in a series. There is one ChartItem instance for each item in the series' 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' ChartItem objects. You can specify a new renderer to use with the series' `itemRenderer` style property. This property points to a class that defines the appearance of the ChartItem object.

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

Chart type	Available renderer classes
AreaChart	AreaRenderer
BarChart BubbleChart ColumnChart PlotChart	BoxItemRenderer CircleItemRenderer CrossItemRenderer DiamondItemRenderer ShadowBoxItemRenderer TriangleItemRenderer
CandlestickChart HLOCChart	CandlestickItemRenderer HLOCItemRenderer
LineChart	LineRenderer 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.

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 CircleItemRenderer. -->
           <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>
```

To apply a renderer to a series in ActionScript, you use the `setStyle()` method. In that method, you create a new ClassFactory and pass the renderer to its constructor. Flex generates an instance of this class to be the renderer. Be sure to import the appropriate classes when using renderer classes.

The following example sets the renderer for the second series to the CircleItemRenderer and the renderer for the third series to the CrossItemRenderer in ActionScript.

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

```
     import mx.charts.renderers.*;

     [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}
     ]);

    private function initSeriesStyles():void {
        // Second series uses CircleItemRenderer.
        series2.setStyle("itemRenderer", new
ClassFactory(mx.charts.renderers.CircleItemRenderer));

        // Third series uses CrossItemRenderer.
        series3.setStyle("itemRenderer", new
ClassFactory(mx.charts.renderers.CrossItemRenderer));
    }

  ]]></mx:Script>
  <mx:Panel title="Plot Chart">
     <mx:PlotChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
     >
        <mx:series>
           <mx:PlotSeries
                id="series1"
                xField="Expenses"
                yField="Profit"
                displayName="Plot 1"
           />

           <mx:PlotSeries
                id="series2"
                xField="Amount"
                yField="Expenses"
                displayName="Plot 2"
           />

           <mx:PlotSeries
                id="series3"
                xField="Profit"
                yField="Amount"
                displayName="Plot 3"
           />
        </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.

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 140.

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 73.

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 706 in *Adobe Flex 3 Developer Guide*.

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(source='../assets/butterfly.gif')"
                radius="20"
```

```
                legendMarkerRenderer="@Embed(source='../assets/butterfly.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 butterfly.gif graphic to represent each data point on the plot chart. It controls the size of the embedded image by using the `radius` style 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, pass it to the ClassFactory class's constructor, 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;
     import mx.core.BitmapAsset;

     [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/butterfly.gif")]
     public var myButterfly:Class;

     [Bindable]
     public var myButterflyFactory:ClassFactory = new ClassFactory(myButterfly);

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

```
            <!-- First series uses custom class renderer. -->
            <mx:PlotSeries
                 id="series1"
                 xField="Expenses"
                 yField="Profit"
                 displayName="Plot 1"
                 itemRenderer="{myButterflyFactory}"
                 legendMarkerRenderer="{myButterflyFactory}"
                 radius="20"
            />

            <!-- Second series uses DiamondItemRenderer. -->
            <mx:PlotSeries
                 id="series2"
                 xField="Amount"
                 yField="Expenses"
                 displayName="Plot 2"
                 itemRenderer="mx.charts.renderers.CircleItemRenderer"
            />

            <!-- Third series uses CrossItemRenderer. -->
            <mx:PlotSeries
                 id="series3"
                 xField="Profit"
                 yField="Amount"
                 displayName="Plot 3"
                 itemRenderer="mx.charts.renderers.CrossItemRenderer"
            />
        </mx:series>
    </mx:PlotChart>
    <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

You can also use the `setStyle()` method to apply the custom class to the item renderer. The following example sets the `itemRenderer` and `legendMarkerRenderer` style properties to the embedded image:

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

     [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/butterfly.gif")]
     public var myButterfly:Class;

    private function setStylesInit():void {
        series1.setStyle("itemRenderer", new ClassFactory(myButterfly));
```

```
        series1.setStyle("legendMarkerRenderer", new ClassFactory(myButterfly));
    }
]]></mx:Script>
<mx:Panel title="Plot Chart">
    <mx:PlotChart id="myChart"
        dataProvider="{expenses}"
        showDataTips="true"
    >
        <mx:series>

            <!-- First series uses custom class renderer. -->
            <mx:PlotSeries
                 id="series1"
                 xField="Expenses"
                 yField="Profit"
                 displayName="Plot 1"
                 radius="20"
            />

            <!-- Second series uses DiamondItemRenderer. -->
            <mx:PlotSeries
                 id="series2"
                 xField="Amount"
                 yField="Expenses"
                 displayName="Plot 2"
                 itemRenderer="mx.charts.renderers.CircleItemRenderer"
            />

            <!-- Third series uses CrossItemRenderer. -->
            <mx:PlotSeries
                 id="series3"
                 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 709 in *Adobe Flex 3 Developer Guide*.

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 title="ColumnChart control with a programmatic ItemRenderer">
     <mx:ColumnChart id="column" dataProvider="{expenses}" showDataTips="true">
```

```
        <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 713 in *Adobe Flex 3 Developer Guide*.

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 205.

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 with chart controls" on page 99.

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}" showDataTips="true">
        <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 with chart controls" on page 99.

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 for the Legend control, 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}" showDataTips="true">
        <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

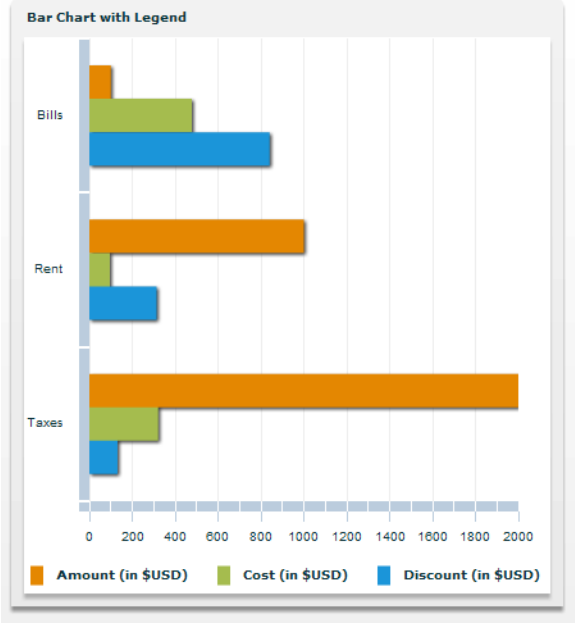
The `direction` property is a commonly used property that is inherited from the Tile container. This property of the `<mx:Legend>` tag causes the LegendItem objects to line up horizontally or vertically. The default value of `direction` is `vertical`; when you use this value, Flex stacks the LegendItem 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}" showDataTips="true">
        <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}
    ]);
 ]]></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' `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 132.

Chapter 4: Displaying Data and Labels

Adobe® Flex® provides some techniques for displaying data and labels in your charts.

Topics

Working with axes. 145
About the CategoryAxis class 145
About the NumericAxis class 147
Adding axis titles. 159
Defining axis labels. 164
Using data labels 171
Using DataTip objects 179
Using per-item fills 190
Using the minField property 196
Stacking charts. 198
Using Legend controls 205

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

For more information about chart data providers, see "Defining chart data" on page 15.

You can also customize the labels using the `labelFunction` property of the CategoryAxis and NumericAxis classes. This property points to a callback function that refines the labels based on the existing data provider. For more information, see "Defining axis labels" on page 164.

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 such as labels and tick marks along the axis.

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 151.

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 164.

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:Panel title="LineChart control with a linear axis">
        <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:Legend dataProvider="{myChart}"/>
    </mx:Panel>
</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:Panel title="LineChart control with a logarithmic axis">
        <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:Legend dataProvider="{myChart}"/>
    </mx:Panel>
</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 object" on page 154.

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 labels 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 151.
- 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 151.
- 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">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection
     [Bindable]
     public var deck: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="Sample DateTimeAxis">
    <mx:LineChart id="myChart"
      dataProvider="{deck}"
      showDataTips="true"
    >
      <mx:horizontalAxis>
        <mx:DateTimeAxis dataUnits="days"/>
      </mx:horizontalAxis>
      <mx:series>
        <mx:LineSeries
            yField="close"
            xField="date"
            displayName="DECK"
        />
      </mx:series>
    </mx:LineChart>
    <mx:Legend dataProvider="{myChart}"/>
  </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:Panel title="DateTimeAxis with parseFunction">
    <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:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</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 164.

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">
  <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 with range">
     <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:Legend dataProvider="{myChart}"/>
  </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 object

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 that 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 includes pricing data only 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 draws only 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 object to `days`. You then use the `disabledDays` or `disabledRanges` properties of the DateTimeAxis object.

The value of the `disabledDays` property of 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">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection
     [Bindable]
     public var deck: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="WorkWeekAxis Example">
     <mx:LineChart id="myChart"
        dataProvider="{deck}"
        showDataTips="true"
     >
        <mx:horizontalAxis>
           <mx:DateTimeAxis
            dataUnits="days"
            parseFunction="myParseFunction"
            disabledDays="{offDays}"
        />
        </mx:horizontalAxis>
        <mx:series>
           <mx:LineSeries
                yField="close"
                xField="date"
                displayName="DECK"
           />
        </mx:series>
     </mx:LineChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

To exclude a range of dates from the DateTimeAxis object, 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 deck: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="Disabled Date Ranges">
   <mx:LineChart id="myChart"
      dataProvider="{deck}"
      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="DECK"
```

```
            />
        </mx:series>
    </mx:LineChart>
    <mx:Legend dataProvider="{myChart}"/>
  </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 deck: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="Disabled date ranges">
    <mx:HBox>
        <mx:LineChart id="myChart"
            dataProvider="{deck}"
            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="DECK"
                />
            </mx:series>
        </mx:LineChart>
        <mx:DateChooser id="dc1"
```

```
            showToday="false"
            click="onDateChange(event)"
            displayedMonth="7"
            displayedYear="2007"
            disabledRanges="{dateChooserDisabledRanges}"
        />
    </mx:HBox>
    <mx:Legend dataProvider="{myChart}"/>

    <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 title="Axis with title">
     <mx:ColumnChart id="myChart"
        showDataTips="true"
        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 title="Axis title with embedded font">
     <mx:ColumnChart id="column"
        showDataTips="true"
        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 656 in *Adobe Flex 3 Developer Guide*.

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 title="Styling Axis Titles with CSS">
     <mx:ColumnChart id="column" dataProvider="{expenses}" showDataTips="true">
        <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"/>
```

To change the appearance of the title on a chart, you can also use the `titleRenderer` property of the AxisRenderer class. This lets you specify a class that defines the appearance of the title. The class must extend UIComponent and implement the IDataRenderer and IFlexDisplayObject interfaces. Its `data` property will be the title used in the chart.

Typically, you extend the ChartLabel class to create a custom title renderer. In that class, you must override the `createChildren()` and `updateDisplayList()` methods. In the `createChildren()` method you create an instance of the TextField class that Flex uses to render the title. In the `updateDisplayList()` method, you set the properties of the TextField object.

The following example is a custom title renderer. It creates a gradient fill background for each of the axis titles.

```
// charts/MyTextRenderer.as
package {
    import mx.charts.chartClasses.ChartLabel;
    import mx.charts.*;
    import flash.display.*;
    import flash.geom.Matrix;
    import flash.text.TextField;

    public class MyTextRenderer extends ChartLabel {

        // The title is renderered in a TextField.
        private var myText:TextField;

        public function MyTextRenderer() {
            super();
        }

        override protected function createChildren():void{
            super.createChildren();
            myText = new TextField();
        }

        override protected function updateDisplayList(w:Number, h:Number):void {
            super.updateDisplayList(w, h);

            // The data property provides access to the title text.
            if(data.hasOwnProperty('text')) {
                myText.text = data.text;
            } else {
                myText.text = data.toString();
            }
```

```
            this.setStyle("textAlign","center");
            var g:Graphics = graphics;
            g.clear();
            var m:Matrix = new Matrix();
            m.createGradientBox(w+100,h,0,0,0);
            g.beginGradientFill(GradientType.LINEAR,[0xFF0000,0xFFFFFF],
                [.1,1],[0,255],m,null,null,0);
            g.drawRect(-50,0,w+100,h);
            g.endFill();
        }
    }
}
```

To use this class in your Flex application, you just point the `titleRenderer` property to it. This assumes that the class is in your source path. In this case, store both the MyTextRenderer.as and RendererExample.mxml files in the same directory.

The following example applies the custom title renderer to all axis titles.

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

        [Bindable]
        private var ac:ArrayCollection;

        public function setupDP():void{
            ac =  new ArrayCollection([
              [ "Multiline\nLabel\nOK", 200000],
              [ "Label 2\nCD\nJK", 150000],
              [ "Label 3\nEF\nLM", 40000]
            ]);
        }
     ]]>
    </mx:Script>

    <mx:HBox>
        <mx:ColumnChart id="bc1"
            axisTitleStyleName="myStyle"
            showDataTips="true"
            dataProvider="{ac}"
        >
            <mx:series>
                <mx:ColumnSeries xField="0" yField="1"/>
            </mx:series>
            <mx:verticalAxis>
                <mx:LinearAxis id="va1" title="values"/>
            </mx:verticalAxis>
            <mx:horizontalAxis >
                <mx:CategoryAxis id="ha1"
                    dataProvider="{ac}"
                    categoryField="0"
                    title="Labels"
                />
            </mx:horizontalAxis>
            <mx:horizontalAxisRenderers>
                <mx:AxisRenderer
```

```
                    axis="{ha1}"
                    canDropLabels="true"
                    titleRenderer="MyTextRenderer"
                />
            </mx:horizontalAxisRenderers>
            <mx:verticalAxisRenderers>
                <mx:AxisRenderer
                    axis="{va1}"
                    canDropLabels="true"
                    titleRenderer="MyTextRenderer"
                    verticalAxisTitleAlignment="vertical"
                />
            </mx:verticalAxisRenderers>
        </mx:ColumnChart>

        <mx:Spacer width="50"/>

        <mx:BarChart id="bc2"
            showDataTips="true"
            dataProvider="{ac}"
        >
            <mx:series>
                <mx:BarSeries xField="1" yField="0"/>
            </mx:series>
            <mx:horizontalAxis>
                <mx:LinearAxis id="ha2" title="Values"/>
            </mx:horizontalAxis>
            <mx:verticalAxis>
                <mx:CategoryAxis id="va2"
                    dataProvider="{ac}"
                    categoryField="0"
                    title="Labels"
                />
            </mx:verticalAxis>
            <mx:horizontalAxisRenderers>
                <mx:AxisRenderer
                    axis="{ha2}"
                    canDropLabels="false"
                    titleRenderer="MyTextRenderer"
                />
            </mx:horizontalAxisRenderers>
            <mx:verticalAxisRenderers>
                <mx:AxisRenderer
                    axis="{va2}"
                    canDropLabels="true"
                    titleRenderer="MyTextRenderer"
                />
            </mx:verticalAxisRenderers>
        </mx:BarChart>
    </mx:HBox>

</mx:Application>
```

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 title="Disabled Axis Labels">
        <mx:ColumnChart id="column" dataProvider="{expenses}" showDataTips="true">
            <mx:horizontalAxis>
                <mx:CategoryAxis
                    id="a1"
                    dataProvider="{expenses}"
                    categoryField="Month"
                />
            </mx:horizontalAxis>

            <mx:horizontalAxisRenderers>
                <mx:AxisRenderer
                    axis="{a1}"
                    showLabels="false"
                />
            </mx:horizontalAxisRenderers>

            <mx:verticalAxisRenderers>
                <mx:AxisRenderer
                    axis="{a1}"
                    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>
```

Note that any time you want to use an AxisRenderer, you must explicitly set the axis to which it is applied with the renderer's `axis` property.

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 signature for a NumericAxis object (including the DateTimeAxis, LinearAxis, and LogAxis classes) is:

```
function_name(labelValue:Object, previousLabelValue:Object, axis:IAxis):return_type
```

The callback function signature for a CategoryAxis object is:

```
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.
	
	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:
	
	`Profit:1500`
	
	`Month:Mar`
	
	`Expenses:500`
	
	You can access the values in this object by using dot-notation for dynamic objects, as the following example shows:
	
	`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 title="Custom Label Function">
   <mx:ColumnChart id="column" dataProvider="{expenses}" showDataTips="true">
      <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 signature for the AxisRenderer's label function is:

```
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' 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 145.

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 66.

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:Panel title="LineChart control with a linear axis">
        <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:Legend dataProvider="{myChart}"/>
    </mx:Panel>
</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.

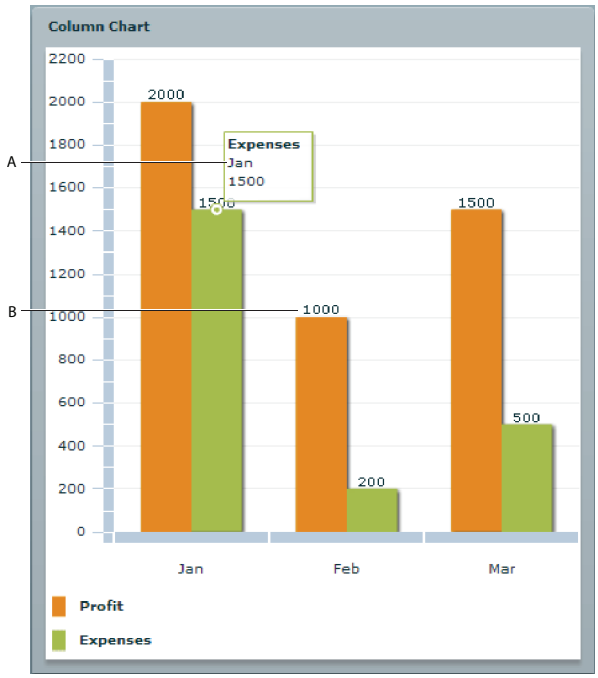
To set the minimum and maximum values on a DateTimeAxis, you must use Date objects rather than Strings or Numbers in the axis's tag. For more information, see "Setting minimum and maximum values on a DateTimeAxis" on page 153.

For information about setting the length and location of tick marks, see "Formatting tick marks" on page 96.

Using data labels

Data 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 DataTip objects in that they typically show only 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. DataTip controls are more interactive, since they appear and disappear as the user moves the mouse over chart elements. For more information about DataTip objects, see "Using DataTip objects" on page 179.

The following example shows a column chart with an active DataTip object and visible data labels:



***A.** DataTip **B.** Data label*

The following chart series support data labels:

- BarSeries
- ColumnSeries
- PieSeries

Just like DataTip controls, data labels are not enabled by default. You can add data labels by setting the value of the series' `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 174. 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 `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:Legend dataProvider="{myChart}"/>
    </mx:Panel>

    <mx:Button id="b1"
      label="Change Label Location"
      click="changeLabelLocation()"
    />

</mx:Application>
```

Adding data labels

By default, BarSeries, ColumnSeries, and PieSeries objects do not display data labels. To enable 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' tag or by using the `setStyle()` method, as the following example shows:

```
mySeries.setStyle("labelPosition", "outside");
```

The contents of the data label are 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 176.
- Default—If neither the `labelField` nor the `labelFunction` are specified, the default content 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 content 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 174.

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:Legend dataProvider="{myChart}"/>
  </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 66.

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 if 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.

ColumnChart controls have two additional properties that you can use to control the way data labels are rendered: `showLabelVertically` and `extendLabelToEnd`. These properties are important when the available space for labels is not sufficient to render the entire label. If you set `showLabelVertically` to `true`, Flex can render the label vertically rather than horizontally if the columns are not wide enough to render them normally. If you set `extendLabelToEnd` to `true`, then Flex can use the space between the data item's edge and the outer boundary of the chart to render the label in, rather than truncate the label at the end of the data item's column.

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 176); for PieSeries objects, the function takes four arguments (see "Customizing data labels for PieSeries objects" on page 178).

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:Legend dataProvider="{myChart}"/>
  </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 item:ColumnSeriesItem = ColumnSeriesItem(element);
var s:String = item.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.
index	The number of the data point in the data provider; type Number.
percentValue	The size of the pie wedge relative to the pie; type Number. If the pie wedge is a quarter of the size of the pie, this value is 25.

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 + "\n$" + 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"
                  nameField="Expense"
            />
         </mx:series>
      </mx:PieChart>
      <mx:Legend dataProvider="{chart}"/>
   </mx:Panel>
</mx:Application>
```

Using DataTip objects

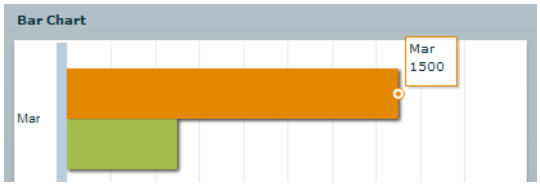
DataTip objects are similar to Flex ToolTip objects in that they cause a small pop-up window to appear that shows the data value for the data point under the mouse pointer.

You use the `showDataTips` property of chart controls to enable DataTip objects.

The values displayed in 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.

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 171*.

The following image shows a simple DataTip:



To enable DataTip objects, 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"
                displayName="Profit"
           />
           <mx:BarSeries
                yField="Month"
                xField="Expenses"
                displayName="Expenses"
           />
        </mx:series>
     </mx:BarChart>
  </mx:Panel>
  <mx:Legend dataProvider="{myChart}"/>
</mx:Application>
```

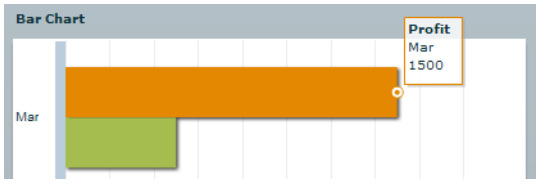
To change the styles of the DataTip, you can use the DataTip type selector in an `<mx:Style>` block or in an external CSS file. The following example applies a new style to the text in the DataTip:

```
<?xml version="1.0"?>
<!-- charts/DataTipStyles.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Style>
        DataTip {
            fontFamily: "Arial";
```

```
            fontSize: 12;
            fontWeight:bold;
            fontStyle:italic;
        }
    </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 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>
```

Customizing the text inside a DataTip object

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 or Adobe® AIR™ displays this name in the DataTip, as the following image shows:

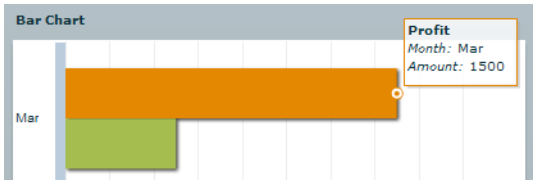


The following example names the data series by using the `displayName` property of the series:

```
<?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>
```

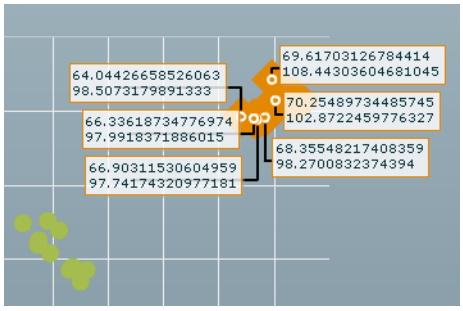
Showing multiple DataTip objects

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 DataTip objects 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"
                     displayName="Profit"
                />
                <mx:BarSeries
                     yField="Month"
                     xField="Expenses"
                     displayName="Expenses"
                />
            </mx:series>
        </mx:BarChart>
        <mx:Legend dataProvider="{myChart}"/>
    </mx:Panel>
</mx:Application>
```

The following example shows DataTip objects 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 DataTip objects 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 DataTip objects 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 219.

You can also show all the available data tips on a chart at one time by setting the value of the chart control's `showAllDataTips` property to `true`. The result is that all data tips are visible at all times. When you do this, you typically set the value of the `showDataTips` property to `false` so that a second data tip does not appear when you mouse over a chart item.

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 which 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 213.

The DataTip callback function can return simple HTML for formatted DataTip objects. Flex supports a small subset of HTML tags including `<FONT>`, `<B>`, `<I>`, and `<BR>`.

Creating custom DataTip renderers

You can create a custom class that defines the appearance and content of the data tip. This class must implement the IFlexDisplayObject and IDataRenderer interfaces.

You typically create a custom ActionScript class that extends DataTip, and override the `createChildren()` and `updateDisplayList()` methods. In the `createChildren()` method, you create an instance of the TextField class. In the `updateDisplayList()` method, you add text to the new text field, and you define the box for the data tip. The chart assigns the custom data tip's `data` property to be the HitData structure for the data point.

The following example class creates a custom data tip.

```
// charts/MyDataTip.as
package {
    import mx.charts.chartClasses.DataTip;
    import mx.charts.*;
    import flash.display.*;
    import flash.geom.Matrix;
    import flash.text.TextField;

    public class MyDataTip extends DataTip {

        // The title is renderered in a TextField.
        private var myText:TextField;

        public function MyDataTip() {
            super();
        }

        override protected function createChildren():void{
            super.createChildren();
            myText = new TextField();
        }

        override protected function updateDisplayList(w:Number, h:Number):void {
            super.updateDisplayList(w, h);
```

```
            // The data property provides access to the data tip's text.
            if(data.hasOwnProperty('text')) {
                myText.text = data.text;
            } else {
                myText.text = data.toString();
            }

            this.setStyle("textAlign","center");
            var g:Graphics = graphics;
            g.clear();
            var m:Matrix = new Matrix();
            m.createGradientBox(w+100,h,0,0,0);
            g.beginGradientFill(GradientType.LINEAR,[0xFF0000,0xFFFFFF],
                [.1,1],[0,255],m,null,null,0);
            g.drawRect(-50,0,w+100,h);
            g.endFill();
        }
    }
}
```

To use the custom data tip, you set the value of the `dataTipRenderer` style property of the chart control to the custom class. The following sample application applies the custom data tip to the chart control.

```
<?xml version="1.0"?>
<!-- charts/CustomDataTipRenderer.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="applyCustomDataTips()">
  <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 applyCustomDataTips():void {
        myChart.setStyle("dataTipRenderer",MyDataTip);
    }

  ]]></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>
```

If you want to prevent the data tip's box from displaying when you mouse over a chart item, you can set the `dataTipRenderer` style property to a dummy class, such as ProgrammaticSkin; for example:

```
myChart.setStyle("dataTipRenderer",mx.skins.ProgrammaticSkin);
```

Using per-item fills

You can customize the appearance of chart items in a series by using the `fillFunction` property to define the fill. This function takes the chart item and its index as arguments, so that you can examine the chart item's data values and return a fill based on whatever criteria you choose to use.

Programmatically assigning fills 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; type ChartItem.
index	The index of the chart item in the series's data provider; type Number.

The `fillFunction` property 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 111.

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` in the data provider when it fills each chart item. If the `yField` value (corresponding to the CurrentAmount field) is greater than $50,000, the column is green. If it is less than $50,000, 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", 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` property to define the fills of chart items, and you want a Legend control in your chart, 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 205.

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", 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 205.

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 gain access to 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 series against the sales goal in the salesGoalSeries series. If the goal is met, the positive difference between the goal and the actual sales is green. If the goal is not yet met, 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", 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 for only the items that have missed sales goals without drawing stacked columns.

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:

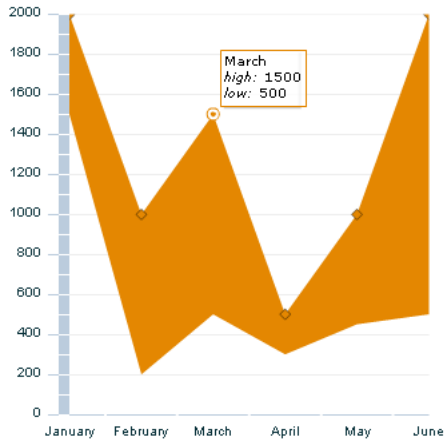
- AreaSeries
- BarSeries
- ColumnSeries

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 id="myChart"
        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:Legend dataProvider="{myChart}"/>
  </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 50.

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

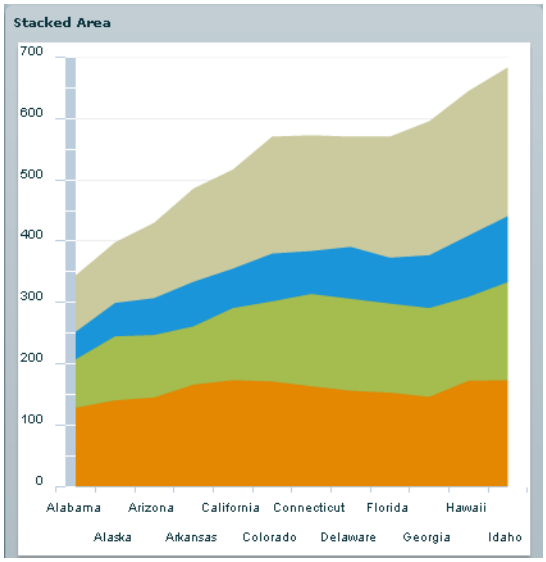
When you use multiple data series in the AreaChart, BarChart, and ColumnChart controls, you can control how Flex displays the series using the `type` property of the controls. The following table describes the values that the `type` property supports:

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.
`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:

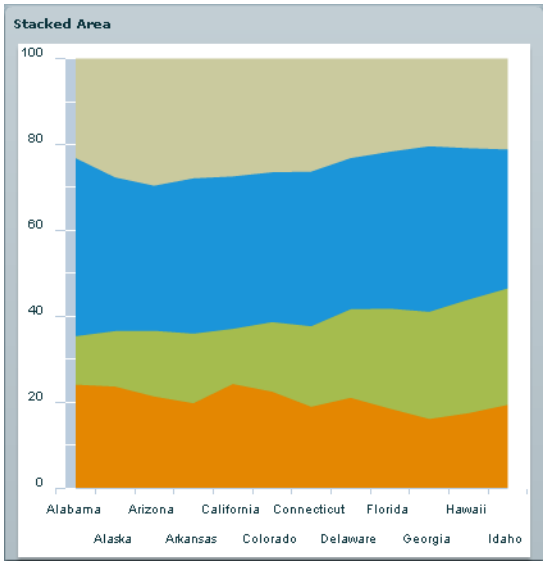
```
<?xml version="1.0"?>
<!-- charts/AreaStacked.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:2000, 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 id="myChart"
        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:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

The following example shows a stacked AreaChart control:



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 105.

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:



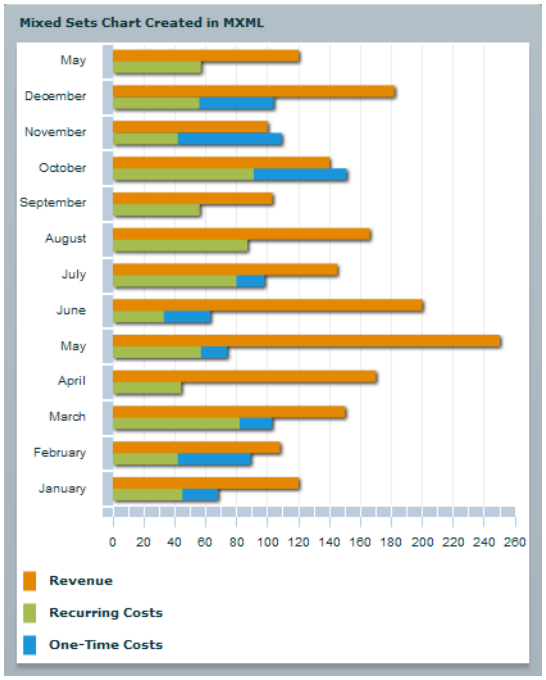
You can use the ColumnSet, BarSet, and AreaSet classes to combine groups of chart series, and thereby use different types of series within the same chart. The following example uses BarSet classes to combine clustered and stacked BarSeries in a single chart. The example shows how to do this in MXML and 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;
        c.showDataTips = true;

        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}" showDataTips="true">
        <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:



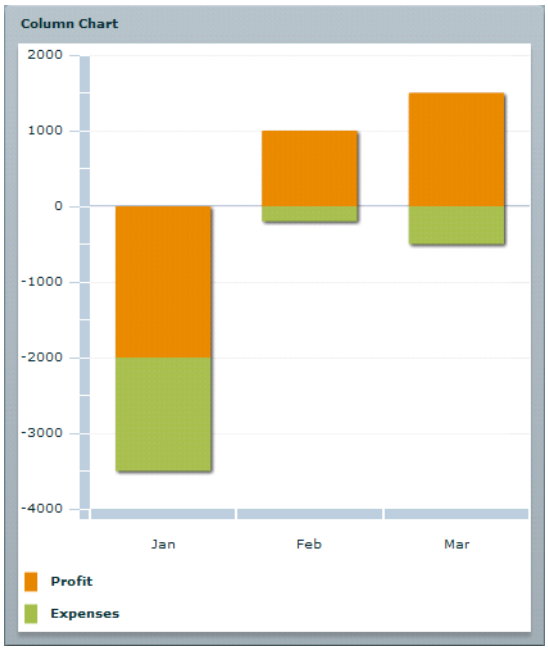
Normally, the values in stacked series are additive, which means that they are all added to one another to create an ever larger data item (column or bar or area). When one or more values is negative, the rendering of the data items can be unpredictable because adding a negative value would normally take away from the data item. You can also use the `allowNegativeForStacked` property of the AreaSet, BarSet, and ColumnSet classes to let a stacked series include data with negative values, and thereby render negative data items that are properly stacked.

The following example stacks the Profit and Expenses fields, in which some of the values are negative.

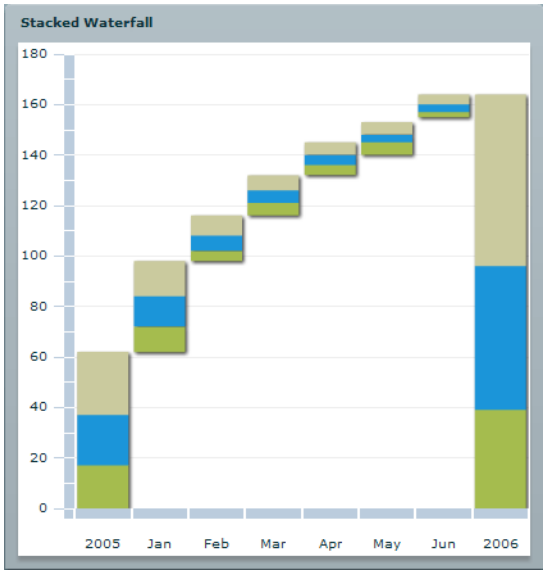
```
<?xml version="1.0"?>
<!-- charts/StackedNegative.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script><![CDATA[
     import mx.collections.ArrayCollection;
     [Bindable]
     public var expenses:ArrayCollection = new ArrayCollection([
        {Month:"Jan", Profit:-2000, 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}" showDataTips="true">
        <mx:horizontalAxis>
           <mx:CategoryAxis
                dataProvider="{expenses}"
                categoryField="Month"
           />
        </mx:horizontalAxis>
        <mx:series>
            <mx:ColumnSet type="stacked" allowNegativeForStacked="true">
                <mx:series>
```

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

The resulting chart shows three months of data, with all expenses and some income data rendering in negative values:



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

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:

A. *Label* ***B.*** *Marker*

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 140.

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}" showDataTips="true">
        <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}" showDataTips="true">
        <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 be 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 DataTip objects 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/MultipleAxesWithCustomLegend.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 5: Using Events and Effects in Charts

You can use Adobe® Flex® charting controls to make interesting and engaging charts. Important factors to consider are how users' interactions with your applications trigger effects and events.

Topics

Handling user interactions with charts 210
Using effects with charts 222
Drilling down into data 235
Selecting chart items 240
Drawing on chart controls 264

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 support for the the ChartEvent and ChartItemEvent events.

The ChartItemEvent class defines events that are specific to the chart components, such as when the user clicks a data item in a chart. The ChartEvent class defines events that occur when the user clicks or double-clicks on a chart control, but not on a chart item in that chart control.

The following table describes the ChartEvent event types:

Chart event type	Description
`chartClick`	Broadcast when the user clicks the mouse button while the mouse pointer is over a chart but not on a chart item.
`chartDoubleClick`	Broadcast when the user double-clicks the mouse button while the mouse pointer is over a chart but not on a chart item.

The following table describes the ChartItemEvent event types:

Chart data event type	Description
`change`	Dispatched when the selected item or items changes in the chart.
`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.
`itemMouseUp`	Broadcast when the user releases the mouse button while over a data point.

Chart data event type	Description
`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.

In addition to the ChartEvent and ChartItemEvent classes, there is also the LegendMouseEvent. This class defines events that are broadcast when the user clicks on legend items or mouses over them.

About chart events

Chart events are triggered when the user clicks or double-clicks the mouse button while the mouse pointer is over a chart control, but not over a chart item in that chart control. These events are dispatched only if the hit data set is empty.

Chart events are of type ChartEvent. Because ChartEvent 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 ChartEvent event.

The following example logs a ChartEvent when you click or double-click the chart control, but only if you are not over a chart item or within it's range, as determined by its `mouseSensitivity` property.

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

    private function chartEventHandler(event:ChartEvent):void {
        /*
        The ChartEvent will only be dispatched if the mouse is _not_ over a
        chart item or if it is outside of the range defined by the
        mouseSensitivity property.
        */
        ta1.text += "Event of type " + event.type + " was triggered.\n";
    }

  ]]></mx:Script>

  <mx:Style>
    Panel {
        paddingLeft:5;
        paddingRight:5;
        paddingTop:5;
        paddingBottom:5;
    }
  </mx:Style>

  <mx:Panel title="Chart click events">
     <mx:HBox>
         <mx:PlotChart id="myChart"
            dataProvider="{expenses}"
```

```
            showDataTips="true"
            mouseSensitivity="10"
            doubleClickEnabled="true"
            chartClick="chartEventHandler(event)"
            chartDoubleClick="chartEventHandler(event)"
        >
            <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:TextArea id="ta1" width="150" height="300"/>
       </mx:HBox>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>

</mx:Application>
```

About chart data events

Chart data events are triggered only when the user moves the mouse pointer over a data point, whereas 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 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" displayName="Expenses"/>
        </mx:series>
     </mx:ColumnChart>
     <mx:Legend dataProvider="{myChart}"/>
  </mx:Panel>
</mx:Application>
```

Using the HitData object

Flex dispatches a ChartItemEvent object for each chart data event such as when you click on an item in the series. 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 219.

The following example uses the `itemDoubleClick` event handler to display HitData information for data points in a column chart when the user clicks on 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 {
        myChart.addEventListener(ChartItemEvent.ITEM_DOUBLE_CLICK, myListener);
```

```
    }
  ]]></mx:Script>
  <mx:Panel title="Accessing HitData Object in Event Handlers">
    <mx:ColumnChart id="myChart"
      dataProvider="{dataSet}"
      doubleClickEnabled="true"
      showDataTips="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:Legend dataProvider="{myChart}"/>
  </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: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:Legend dataProvider="{myChart}"/>
</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 132.

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, to 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 216. The `localToData()` method returns an Array of the data. For more information, see "Using the localToData() method" on page 217.

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 219.

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)"
        showDataTips="true"
     >
        <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 ChartItemEvent (of types such as `itemRollOver` or `itemClick`). Flex still responds to events such as `mouseOver` and `click` by generating a ChartEvent object of type `chartClick` or `chartDoubleClick`.

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="myChart"
        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:HBox>
         <mx:Legend dataProvider="{myChart}"/>
         <mx:VBox>
             <mx:Label text="Mouse Sensitivity:"/>
             <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:VBox>
      </mx:HBox>
  </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 DataTip objects" on page 183.

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="myChart"
        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:Legend dataProvider="{myChart}"/>
  </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 charting controls also include the `showDataEffect` and `hideDataEffect` triggers.

For information on creating complex effects, see "Using Behaviors" on page 545 in *Adobe Flex 3 Developer Guide*.

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 223).

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:

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 the SeriesSlide effect making 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 control's 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 var year:int = 1;

        public function changeProvider():void {
            if (year == 2) {
                myChart.dataProvider=expenses1;
                b1.label="View Second Year Data";
                lbl.text="First Year Data";
                year=1;
            } else {
                myChart.dataProvider=expenses2;
                lbl.text="Second Year Data";
                b1.label="View First Year Data";
                year=2;
            }
        }
    ]]></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"
```

```
    layout="absolute" height="493">
        <mx:Label id="lbl"
            text="First Year Data"
            width="233.5"
            height="44"
            fontSize="24"
            color="#091D96"
            top="5"
            right="10"
        />
        <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}" bottom="0"/>
    </mx:Panel>
    <mx:Button id="b1" click="changeProvider()" label="View Second Year Data"/>
</mx:Application>
```

This example explicitly defines the minimum and maximum values of the vertical axis. If 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 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 30.

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. The `duration` property acts as a minimum duration. The effect can take longer based on the settings of other properties. The default value is 500.
`elementOffset`	The amount of time, in milliseconds, that Flex delays the effect on each element in the series. Set the `elementOffset` property to 0 to make all elements start at the same time and end at the same time. 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. Set the `elementOffset` property to a negative value to make the effect display the last element in the series first. The default value is 20.
`minimumElementDuration`	The amount of time, in milliseconds, that an individual element should take to complete the effect. Charts with a variable number of data points in the series cannot reliably create smooth effects with only the `duration` property. 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. 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. 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`. The default value is 0, which means that the `duration` property is used to control the total duration of the effect.
`offset`	The amount of time, in milliseconds, that Flex delays the start of the effect. Use this property to stagger effects on multiple series. 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 on the screen 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 230.

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`	Defines the location of the focal point of the zoom.
`verticalFocus`	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). Valid values of the `horizontalFocus` property are `left`, `center`, `right`, and `undefined`. Valid values of the `verticalFocus` property are `top`, `center`, `bottom`, and `undefined`. 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. 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`. 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. 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 230.

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 "Using Styles and Themes" on page 589 in *Adobe Flex 3 Developer Guide*.

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"
            />
        </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 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 {
        rearrangeData = new SeriesInterpolate();

        // 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 title="Effects">
   <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 1247 in *Adobe Flex 3 Developer Guide*.

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 10.

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 213.

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 240.

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' 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' 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 243.
- Keyboard selection—Use the keyboard to select one or more data points, as described in "Keyboard and mouse selection" on page 243.
- 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 243.
- Programmatic selection—Programmatically select one or more data points using the chart selection API. For more information, see "Programmatic selection" on page 244.

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, 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>
```

You can specify the colors that are used to indicate if an item is selected, disabled, or being rolled over. You do this by using the `itemSelectionColor`, `itemDisabledColor`, and `itemRollOverColor` style properties of the chart controls. The following example specifies different colors for these states for several chart types:

```
BarChart, PieChart, PlotChart {
    itemSelectionColor: red;
    itemDisabledColor: green;
    itemRollOverColor: blue;
}
```

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' `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 -= 1;
        }

        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 249.

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 is 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' `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 244.

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 244. 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`, `focusedSelected`, 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 `getItemsInRegion()` method to define a rectangular area and select those chart items that are within that region. The `getItemsInRegion()` method takes an instance of the Rectangle class as its only argument. This rectangle defines an area on the stage, in global coordinates.

Getting an array of ChartItem objects with the `getItemsInRegion()` 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 `getItemsInRegion()` 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 area.
            var a:Array = myChart.getItemsInRegion(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, use the following syntax.

Access the properties of the series item

```
var csi:ColumnSeriesItem = ColumnSeriesItem(a[i]);
trace(csi.yValue);
```

Access the item's fields

```
var csi:ColumnSeriesItem = ColumnSeriesItem(a[i]);
trace(csi.item.Profit);
```

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 the 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 `clearSelection()` 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.clearSelection();
            }
        }
    ]]>
    </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 243.

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;
        import mx.charts.Legend;

        [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);

        // Create a legend for the new chart.
        var newLegend:Legend = new Legend();
        newLegend.dataProvider = newChart;

        p1.addChild(newChart);
        p1.addChild(newLegend);
    }

    private function createNewChart(e:Event):void {
        newChart.dataProvider = getNewDataProvider();
    }
]]>
</mx:Script>

<mx:SeriesInterpolate id="interpol"
    duration="1000"
    elementOffset="0"
    minimumElementDuration="200"
/>

<mx:Panel 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: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.charts.Legend;

        [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);

    // Create a legend for the new chart.
    var newLegend:Legend = new Legend();
    newLegend.dataProvider = newChart;

    p2.addChild(newChart);
    p2.addChild(newLegend);
}

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 "Using Drag and Drop" on page 741 in *Adobe Flex 3 Developer Guide*.

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 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/DragDropCustomDragImage.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" creationComplete="initApp()"
xmlns:local="*">
    <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;
        import mx.charts.Legend;

        [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);

    // Create a legend for the new chart.
    var newLegend:Legend = new Legend();
    newLegend.dataProvider = newChart;

    p2.addChild(newChart);
    p2.addChild(newLegend);
}

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) class or the mx.charts.chartClasses.PolarDataCanvas (for polar charts) classes. You can either attach visual controls to the canvas or you can draw on the canvas by 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. As with 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' `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' `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 does not
    // 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"
        showDataTips="true"
        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:HBox>
        <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:HBox>

     <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>
```

You can access an array of the data children by using the `dataChildren` property of the canvas. This property is an Array of the child objects on the canvas.

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, 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, as the following example illustrates:

```
<ColumnChart width="100%" height="100%" creationComplete="createLabel();">
    <annotationElements>
        <CartesianDataCanvas id="canvas"
            includeInRanges="true"
            horizontalAxis={h1}
            verticalAxis={v1}/>
    </annotationElements>
    ...
</ColumnChart>
```

Part 2: Advanced Data Grid Controls and Automation Tools

Topics

Creating OLAP Data Grids 322
Creating Custom Agents 381
Creating Applications for Testing 356

Chapter 6: Using the AdvancedDataGrid Control

The AdvancedDataGrid control expands on the functionality of the standard DataGrid control to add data visualization capabilities to your Adobe® Flex™ application. These capabilities provide greater control of data display, data aggregation, and data formatting.

For more information on the DataGrid control, see "DataGrid control" on page 395.

Topics

About the AdvancedDataGrid control 275
Sorting by multiple columns 278
Styling rows and columns 280
Selecting multiple cells and rows 284
Hierarchical and grouped data display 287
Displaying hierarchical data 294
Displaying grouped data 296
Creating column groups 309
Using item renderers with the AdvancedDataGrid control 313
Keyboard navigation 320

About the AdvancedDataGrid control

The AdvancedDataGrid control extends the capabilities of the standard DataGrid control to improve data visualization. The following table describes the main data visualization capabilities of the AdvancedDataGrid control:

Capability	Description
Sorting by multiple columns	Sort by multiple columns when you click in the column header. For more information, see "Sorting by multiple columns" on page 278.
Styling rows and columns	Use the `styleFunction` property to specify a function to apply styles to rows and columns of the control. For more information, see "Styling rows and columns" on page 280.
Displaying hierarchical and grouped data	Use an expandable navigation tree in a column to control the visible rows of the control. For more information, see "Hierarchical and grouped data display" on page 287.
Creating column groups	Collect multiple columns under a single column heading. For more information, see "Creating column groups" on page 309.
Using item renderers	Span multiple columns with an item renderer and use multiple item renderers in the same column. For more information, see "Using item renderers with the AdvancedDataGrid control" on page 313.

Displaying hierarchical and grouped data

One of the most important aspects of the AdvancedDataGrid control is its support for the display of hierarchical and grouped data. *Hierarchical data* is data already in a structure of parent and child data items. *Grouped data* is flat data with no inherent hierarchy. Before passing flat data to the AdvancedDataGrid control, you specify one or more data fields that are used to group the flat data into a hierarchy.

To support the display of hierarchical and grouped data, the AdvancedDataGrid control displays a navigation tree in a column that lets you navigate the data hierarchy. The following example shows an AdvancedDataGrid control with a navigation tree in the first column that controls the visible rows of the control, but it can be in any column:

Migrating from the DataGrid control

Besides the major new features added to the AdvancedDataGrid control described in the section "About the AdvancedDataGrid control" on page 275, the AdvancedDataGrid control adds the following property:

firstVisibleItem Specifies the item that is currently displayed in the top row of the AdvancedDataGrid control.

For a complete list of properties and methods of the control, see the *Adobe Flex Language Reference*.

One consideration when migrating an existing application from the DataGrid control to the AdvancedDataGrid control is that the data type of many event objects dispatched by the AdvancedDataGrid control is of type AdvancedDataGridEvent, not of type DataGridEvent as it is for the DataGrid control. If your application references the event object by type, you must update your application so that it uses the correct data type.

To determine the type of the event object for any event, see the *Adobe Flex Language Reference*.

Example: Creating column groups

Column grouping lets you collect multiple columns under a single column heading. The following example shows the Actual and Estimate columns grouped under the Revenues column:

Region	Territory	Territory Rep	Revenues	
			Actual	Estimate
Southwest	Arizona	Barbara Jennings	38865	40000
Southwest	Arizona	Dana Binn	29885	30000
Southwest	Central California	Joe Smith	29134	30000
Southwest	Nevada	Bethany Pittman	52888	45000
Southwest	Northern California	Lauren Ipsum	38805	40000
Southwest	Northern California	T.R. Smith	55498	40000
Southwest	Southern California	Alice Treu	44985	45000
Southwest	Southern California	Jane Grove	44913	45000

The AdvancedDataGrid control is defined by the mx.controls.AdvancedDataGrid class and additional classes in the package mx.controls.advancedDataGridClasses. This package includes the AdvancedDataGridColumn class that you use to define a column.

The following code creates the column grouping shown in the previous image:

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

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            // Import the data used by the AdvancedDataGrid control.
            include "SimpleFlatData.as";
        ]]>
    </mx:Script>

    <mx:AdvancedDataGrid id="myADG"
        dataProvider="{dpFlat}"
        width="100%" height="100%">
        <mx:groupedColumns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumnGroup headerText="Revenues">
                <mx:AdvancedDataGridColumn dataField="Actual"/>
                <mx:AdvancedDataGridColumn dataField="Estimate"/>
            </mx:AdvancedDataGridColumnGroup>
        </mx:groupedColumns>
   </mx:AdvancedDataGrid>
</mx:Application>
```

This example imports the data used by the AdvancedDataGrid control from a file named SimpleFlatData.as. You can see the contents of that file in the topic "Hierarchical and grouped data display" on page 287.

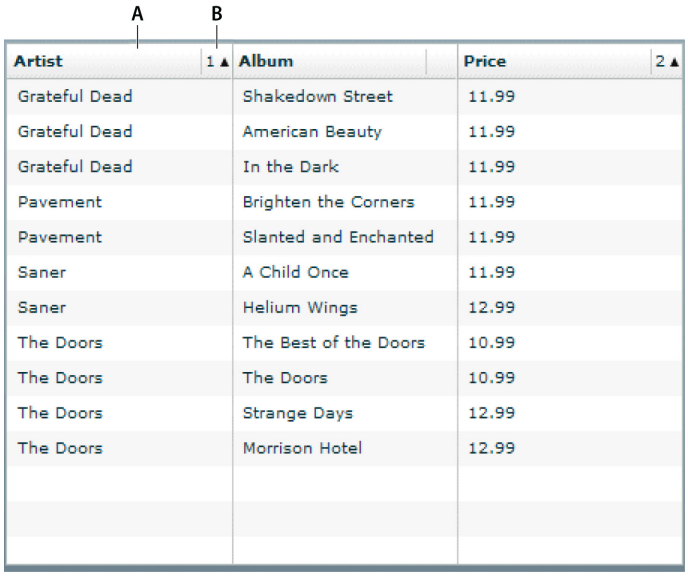
Sorting by multiple columns

By default, the AdvancedDataGrid control displays data in the order specified in the data passed to its `dataProvider` property. The AdvancedDataGrid control also lets you sort data after the control displays it in a single column or multiple columns.

To disable sorting for an entire AdvancedDataGrid control, set the `AdvancedDataGrid.sortableColumns` property to `false`. To disable sorting for an individual column, set the `AdvancedDataGridColumn.sortable` property to `false`.

The way that you sort multiple columns is based on the setting of the `sortExpertMode` property. By default, the `sortExpertMode` property is set to `false`. This setting means that you click in the header area of a column to sort the rows of the AdvancedDataGrid control by that column. Then you click in the multiple column sort area of the header to sort by additional columns. To use the Control key to select every column after the first column to perform sort, set the `sortExpertMode` property to `true`.

The following example shows an AdvancedDataGrid control with three columns with the `sortExpertMode` property set to `false`:



***A.** Column header area **B.** Multiple column sort area*

Sort the columns with sortExpertMode set to false

1 Click in the column header area of any column in the AdvancedDataGrid control to sort by that column. For example, click in the column header for the Artist column to sort that column in ascending order. Click the Artist column header again to sort in descending order.

2 Click in the multiple column sort area of any other column header. For example, click in the multiple column sort area of the Price column to arrange it in ascending order while keeping the Artist column sorted in descending order. You can now find the least expensive album for each artist.

3 Click in the multiple column sort area for the Price column again to arrange it in descending order.

4 Click in the multiple column sort area for any other column header to include other columns in the sort.

Sort the columns with sortExpertMode set to true

1 Click in the column header area of any column in the AdvancedDataGrid control to sort by that column. For example, click in the column header for the Artist column to sort that column in ascending order. Click the Artist column header again to sort in descending order.

2 While holding down the Control key, click in any other column header. For example, click the column header for the Price column to arrange it in ascending order while keeping the Artist column sorted in descending order. You can now find the least expensive album for each artist.

3 While holding down the Control key, click the column header for the Price column again to arrange it in descending order.

4 While holding down the Control key, select any other column header to include other columns in the sort.

The following code implements multiple column sort when the `sortExpertMode` property is set to `true`:

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

    <mx:Script>
      <![CDATA[
         import mx.collections.ArrayCollection;

         [Bindable]
         private var dpADG:ArrayCollection = new ArrayCollection([
           {Artist:'Pavement', Album:'Slanted and Enchanted', Price:11.99},
           {Artist:'Pavement', Album:'Brighten the Corners', Price:11.99},
           {Artist:'Saner', Album:'A Child Once', Price:11.99},
           {Artist:'Saner', Album:'Helium Wings', Price:12.99},
           {Artist:'The Doors', Album:'The Doors', Price:10.99},
           {Artist:'The Doors', Album:'Morrison Hotel', Price:12.99},
           {Artist:'Grateful Dead', Album:'American Beauty', Price:11.99},
           {Artist:'Grateful Dead', Album:'In the Dark', Price:11.99},
           {Artist:'Grateful Dead', Album:'Shakedown Street', Price:11.99},
           {Artist:'The Doors', Album:'Strange Days', Price:12.99},
           {Artist:'The Doors', Album:'The Best of the Doors', Price:10.99}
         ]);
      ]]>
    </mx:Script>

    <mx:AdvancedDataGrid
        width="100%" height="100%"
        sortExpertMode="true"
        dataProvider="{dpADG}">
        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Artist" />
            <mx:AdvancedDataGridColumn dataField="Album" />
            <mx:AdvancedDataGridColumn dataField="Price" />
        </mx:columns>
   </mx:AdvancedDataGrid>
</mx:Application>
```

Styling rows and columns

You apply styles to the rows and columns of the AdvancedDataGrid control by using callback functions. To control the styling of a row, use the `AdvancedDataGrid.styleFunction` property to specify the callback function; to control the styling of a column, use the `AdvancedDataGridColumn.styleFunction` property to specify the callback function.

The callback function specified by the `AdvancedDataGrid.styleFunction` property is invoked first, followed by the function specified by the `AdvancedDataGridColumn.styleFunction` property. Therefore, the AdvancedDataGrid control applies styles to rows first, and then to columns.

The callback function must have the following signature:

```
function_name(data:Object, column:AdvancedDataGridColumn):Object
```

The function returns an Object that contains one or more *styleName:value* pairs to specify a style setting, or null. The *styleName* field contains the name of a style property, such as `color`, and the *value* field contains the value for the style property, such as 0x00FF00. For example, you could return two styles using the following code

```
{color:0xFF0000, fontWeight:"bold"}
```

The AdvancedDataGrid control invokes the callback function when it updates its display, such as when the control is first drawn on application start up, or when you call the `invalidateList()` method.

The examples in this section use callback functions to set the style of an AdvancedDataGrid control. All of the examples in this section use the following data from the StyleData.as file:

```
[Bindable]
private var dpADG:ArrayCollection = new ArrayCollection([
    {Artist:'Pavement', Album:'Slanted and Enchanted', Price:11.99},
    {Artist:'Pavement', Album:'Brighten the Corners', Price:11.99},
    {Artist:'Saner', Album:'A Child Once', Price:11.99},
    {Artist:'Saner', Album:'Helium Wings', Price:12.99},
    {Artist:'The Doors', Album:'The Doors', Price:10.99},
    {Artist:'The Doors', Album:'Morrison Hotel', Price:12.99},
    {Artist:'Grateful Dead', Album:'American Beauty', Price:11.99},
    {Artist:'Grateful Dead', Album:'In the Dark', Price:11.99},
    {Artist:'Grateful Dead', Album:'Shakedown Street', Price:11.99},
    {Artist:'The Doors', Album:'Strange Days', Price:12.99},
    {Artist:'The Doors', Album:'The Best of the Doors', Price:10.99}
]);
```

Styling rows

The following example uses the `AdvancedDataGrid.styleFunction` property to specify a callback function to apply styles to the rows of an AdvancedDataGrid control. In this example, you use Button controls to select an artist in the AdvancedDataGrid control. Then the callback function highlights in red all rows for the selected artist.

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

    <mx:Script>
      <![CDATA[
          import mx.collections.ArrayCollection;
          import mx.controls.advancedDataGridClasses.AdvancedDataGridColumn;

          // Include the data for the AdvancedDataGrid control.
          include "StyleData.as"

          // Artist name to highlight.
```

```
            protected var artistName:String;

            // Event handler to set the selected artist's name
            // based on the selected Button control.
            public function setArtistName(event:Event):void
            {
               artistName=Button(event.currentTarget).label;
               // Refresh row display.
               myADG.invalidateList();
            }

            // Callback function that highlights in red
            // all rows for the selected artist.
            public function myStyleFunc(data:Object,
              col:AdvancedDataGridColumn):Object
            {
               if (data["Artist"] == artistName)
                  return {color:0xFF0000};

               // Return null if the Artist name does not match.
               return null;
            }
        ]]>
    </mx:Script>

    <mx:AdvancedDataGrid id="myADG"
        width="100%" height="100%"
        dataProvider="{dpADG}"
        styleFunction="myStyleFunc">
        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Artist"/>
            <mx:AdvancedDataGridColumn dataField="Album"/>
            <mx:AdvancedDataGridColumn dataField="Price"/>
        </mx:columns>
    </mx:AdvancedDataGrid>

    <mx:HBox>
       <mx:Button label="Pavement" click="setArtistName(event);"/>
       <mx:Button label="Saner" click="setArtistName(event);"/>
       <mx:Button label="The Doors" click="setArtistName(event);"/>
    </mx:HBox>
</mx:Application>
```

Styling columns

The following example modifies the example in the previous section to use a callback function to apply styles to the rows and one column of an AdvancedDataGrid control. In this example, the Price column displays all cells with a price less than $11.00 in green:

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

    <mx:Script>
      <![CDATA[
          import mx.collections.ArrayCollection;
          import mx.controls.advancedDataGridClasses.AdvancedDataGridColumn;

          // Include the data for the AdvancedDataGrid control.
          include "StyleData.as"
```

```
        // Artist name to highlight.
        protected var artistName:String;

        // Event handler to set the selected artist's name
        // based on the selected Button control.
        public function setArtistName(event:Event):void
        {
           artistName=Button(event.currentTarget).label;
           // Refresh row display.
           myADG.invalidateList();
        }

        // Callback function that highlights in red
        // all rows for the selected artist.
        public function myStyleFunc(data:Object,
          col:AdvancedDataGridColumn):Object
        {
           if (data["Artist"] == artistName)
              return {color:0xFF0000};

           // Return null if the Artist name does not match.
           return null;
        }

       // Callback function that hightlights in green
       // all entries in the Price column
       // with a value less than $11.00.
       public function myColStyleFunc(data:Object,
          col:AdvancedDataGridColumn):Object
       {
           if(data["Price"] <= 11.00)
               return {color:0x00FF00};

           return null;
       }
     ]]>
   </mx:Script>

   <mx:AdvancedDataGrid id="myADG"
       width="100%" height="100%"
       dataProvider="{dpADG}"
       styleFunction="myStyleFunc">
       <mx:columns>
           <mx:AdvancedDataGridColumn dataField="Artist"/>
           <mx:AdvancedDataGridColumn dataField="Album"/>
           <mx:AdvancedDataGridColumn dataField="Price"
               styleFunction="myColStyleFunc"/>
       </mx:columns>
   </mx:AdvancedDataGrid>

   <mx:HBox>
      <mx:Button label="Pavement" click="setArtistName(event);"/>
      <mx:Button label="Saner" click="setArtistName(event);"/>
      <mx:Button label="The Doors" click="setArtistName(event);"/>
   </mx:HBox>
</mx:Application>
```

Using a data formatter in a column

The AdvancedDataGridColumn class contains a `formatter` property that lets you pass an instance of the Formatter class to the column, or an instance of a subclass of the Formatter class. The formatter classes convert data into customized strings. For example, you can use the CurrencyFormatter class to prefix the value in the Price column with a dollar ($) sign.

You use a formatter class with a callback function specified by the `labelFunction` property or the `styleFunction` property. If you specify a callback function, Flex invokes that formatter class after invoking the callback functions.

The following example modifies the example from the section "Styling rows" on page 280 to add a CurrencyFormatter class to the Price column to prefix the price with the $ sign:

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

    <mx:Script>
      <![CDATA[
          import mx.collections.ArrayCollection;
          import mx.controls.advancedDataGridClasses.AdvancedDataGridColumn;

          // Include the data for the AdvancedDataGrid control.
          include "StyleData.as"

          // Artist name to highlight.
          protected var artistName:String;

          // Event handler to set the selected artist's name
          // based on the selected Button control.
          public function setArtistName(event:Event):void
          {
             artistName=Button(event.currentTarget).label;
             // Refresh row display.
             myADG.invalidateList();
          }

          // Callback function that hightlights in red
          // all rows for the selected artist.
          public function myStyleFunc(data:Object,
            col:AdvancedDataGridColumn):Object
          {
             if (data["Artist"] == artistName)
                return {color:0xFF0000};

             // Return null if the Artist name does not match.
             return null;
          }
      ]]>
    </mx:Script>

    <mx:AdvancedDataGrid id="myADG"
        width="100%" height="100%"
        dataProvider="{dpADG}"
        styleFunction="myStyleFunc">
        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Artist"/>
            <mx:AdvancedDataGridColumn dataField="Album"/>
            <mx:AdvancedDataGridColumn width="75" dataField="Price">
                <mx:formatter>
                    <mx:CurrencyFormatter/>
```

```
                </mx:formatter>
            </mx:AdvancedDataGridColumn>
        </mx:columns>
    </mx:AdvancedDataGrid>

    <mx:HBox>
       <mx:Button label="Pavement" click="setArtistName(event);"/>
       <mx:Button label="Saner" click="setArtistName(event);"/>
       <mx:Button label="The Doors" click="setArtistName(event);"/>
    </mx:HBox>
</mx:Application>
```

Selecting multiple cells and rows

All list-based controls support the `allowMultipleSelection` property. Setting the `allowMultipleSelection` property to `true` lets you select more than one item in the control at the same time. For example, the DataGrid control lets you select multiple rows so that you can drag and drop them to another DataGrid control.

The AdvancedDataGrid control adds the capability of letting you select multiple cells. You can drag the selected cells to another AdvancedDataGrid control, copy them to the clipboard, or perform some other action on the cell data.

You use the `selectionMode` property of the AdvancedDataGrid control, with the `allowMultipleSelection` property to configure multiple selection. The default value of the `selectionMode` property is `singleRow`, which means that you can select only a single row at a time. You can also set the `selectionMode` property to `singleCell`, `multipleRows` or to `multipleCells`.

Select contiguous items in the control

1. Click the first item, either a row or cell, to select it.
2. Hold down the Shift key as you select an additional item.
   - If the `selectionMode` property is set to `multipleRows`, click any cell in another row to select multiple, contiguous rows.
   - If the `selectionMode` property is set to `multipleCells`, click any cell to select multiple, contiguous cells.

Select discontiguous items in the control

1. Click the first item, either a row or cell, to select it.
2. Hold down the Control key as you select an additional item.
   - If the `selectionMode` property is set to `multipleRows`, click any cell in another row to select that single row.
   - If the `selectionMode` property is set to `multipleCells`, click any cell to select that single cell.

As you select cells or rows in the AdvancedDataGrid control, the control updates the `selectedCells` property with information about your selection. The `selectedCells` property is an Array of Objects where each Object contains a `rowIndex` and `columnIndex` property that represents the location of the selected row or cell in the control.

The value of the `selectionMode` property determines the data in the `rowIndex` and `columnIndex` properties, as the following tables describes:

Value of selectionMode property	**Value of rowIndex and columnIndex properties**
`none`	No selection allowed in the control, and `selectedCells` is null.
`singleRow`	Click any cell in the row to select the row. After the selection, `selectedCells` contains a single Object: `[{rowIndex:`*`selectedRowIndex`*`, columnIndex: -1}]`

`multipleRows`	Click any cell in the row to select the row.
	• While holding down the Control key, click any cell in another row to select the row for discontiguous selection.
	• While holding down the Shift key, click any cell in another row to select multiple, contiguous rows.
	After the selection, `selectedCells` contains one Object for each selected row:
	`[ {rowIndex: `*`selectedRowIndex1`*`, columnIndex: -1},`
	`{rowIndex: `*`selectedRowIndex2`*`, columnIndex: -1},`
	`...`
	`{rowIndex: `*`selectedRowIndexN`*`, columnIndex: -1}`
	`]`

`singleCell`	Click any cell to select the cell. After the selection, `selectedCells` contains a single Object: `[{rowIndex: `*`selectedRowIndex`*`, columnIndex:`*`selectedColIndex`*`}]`

`multipleCells`	Click any cell to select the cell.
	• While holding down the Control key, click any cell to select the cell multiple discontiguous selection.
	• While holding down the Shift key, click any cell to select multiple, contiguous cells.
	After the selection, `selectedCells` contains one Object for each selected cell:
	`[ {rowIndex: `*`selectedRowIndex1`*`, columnIndex: `*`selectedColIndex1`*`},`
	`{rowIndex: `*`selectedRowIndex2`*`, columnIndex: `*`selectedColIndex2`*`},`
	`...`
	`{rowIndex: `*`selectedRowIndexN`*`, columnIndex: `*`selectedColIndexN`*`}`
	`]`

The following example sets the `selectionMode` property to `multipleCells` to let you select multiple cells in the control. This application uses an event handler for the `keyUp` event to recognize the Control+C key combination and, if detected, copies the selected cells from the AdvancedDataGrid control to your system's clipboard.

After you copy the cells, you can paste the cells in another location in the Flex application, or paste them in another application, such as Microsoft Excel. In this example, you paste them to the TextArea control located at the bottom of the application:

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

    <mx:Script>
      <![CDATA[
        import mx.collections.ArrayCollection;
        import flash.events.KeyboardEvent;
        import flash.system.System;

        include "StyleData.as"
```

```
        // Event handler to recognize when Ctrl-C is pressed,
        // and copy the selected cells to the system clipboard.
        private function myKeyUpHandler(event:KeyboardEvent):void
        {
          var keycode_c:uint = 67;
          if (event.ctrlKey && event.keyCode == keycode_c)
          {
              // Separator used between Strings sent to clipboard
              // to separate selected cells.
              var separator:String = ",";
              // The String sent to the clipboard
              var dataString:String = "";

              // Loop over the selectedCells property.
              // Data in selectedCells is ordered so that
              // the last selected cell is at the head of the list.
              // Process the data in reverse so
              // that it appears in the correct order in the clipboard.
              var n:int = event.currentTarget.selectedCells.length;
              for (var i:int = 0; i < n; i++)
              {
                  var cell:Object = event.currentTarget.selectedCells[i];

                  // Get the row for the selected cell.
                  var data:Object =
                    event.currentTarget.dataProvider[cell.rowIndex];

                  // Get the name of the field for the selected cell.
                  var dataField:String =
                    event.currentTarget.columns[cell.columnIndex].dataField;

                  // Get the cell data using the field name.
                  dataString = data[dataField] + separator + dataString;
              }

              // Remove trailing separator.
              dataString =
                dataString.substr(0, dataString.length - separator.length);

              // Write dataString to the clipboard.
              System.setClipboard(dataString);
          }
        }
     ]]>
    </mx:Script>

    <mx:AdvancedDataGrid width="100%" height="100%"
        dataProvider="{dpADG}"
        selectionMode="multipleCells"
        allowMultipleSelection="true"
        keyUp="myKeyUpHandler(event);">
        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Artist"/>
            <mx:AdvancedDataGridColumn dataField="Album"/>
            <mx:AdvancedDataGridColumn dataField="Price"/>
        </mx:columns>
   </mx:AdvancedDataGrid>

   <mx:TextArea id="myTA"/>
</mx:Application>
```

Hierarchical and grouped data display

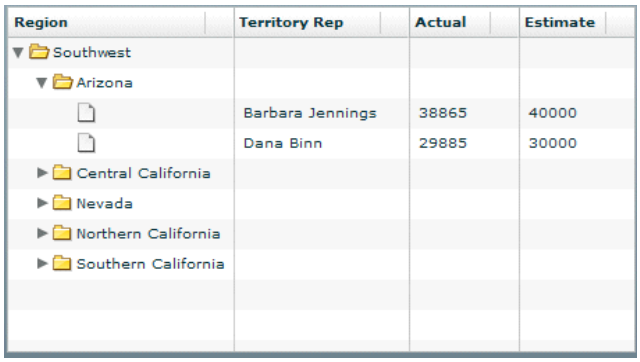
The AdvancedDataGrid control supports the display of hierarchical and grouped data. To support this display, the AdvancedDataGrid control displays a navigation tree in a column that lets you navigate the data hierarchy.

Hierarchical data is data already in a structure of parent and child data items. You can pass hierarchical data directly to the AdvancedDataGrid control. *Grouped data* is data that starts out as flat data with no inherent hierarchy. Before passing the flat data to the AdvancedDataGrid control, you specify one or more data fields that are used to group the flat data into a hierarchy.

The following code shows hierarchical data in the SimpleHierarchicalData.as file:

```
[Bindable]
private var dpHierarchy:ArrayCollection = new ArrayCollection([
  {Region:"Southwest", children: [
     {Region:"Arizona", children: [
        {Territory_Rep:"Barbara Jennings", Actual:38865, Estimate:40000},
        {Territory_Rep:"Dana Binn", Actual:29885, Estimate:30000}]},
     {Region:"Central California", children: [
        {Territory_Rep:"Joe Smith", Actual:29134, Estimate:30000}]},
     {Region:"Nevada", children: [
        {Territory_Rep:"Bethany Pittman", Actual:52888, Estimate:45000}]},
     {Region:"Northern California", children: [
        {Territory_Rep:"Lauren Ipsum", Actual:38805, Estimate:40000},
        {Territory_Rep:"T.R. Smith", Actual:55498, Estimate:40000}]},
     {Region:"Southern California", children: [
        {Territory_Rep:"Alice Treu", Actual:44985, Estimate:45000},
        {Territory_Rep:"Jane Grove", Actual:44913, Estimate:45000}]}
  ]}
]);
```

Notice that the data contains a top-level object that contains a Region field and multiple second-level children. Each second-level child also contains a Region field and one or more additional children. The following example shows the AdvancedDataGrid control displaying this data:



The code for this example is in the section "Controlling the navigation tree of the AdvancedDataGrid control" on page 289.

To display flat data as a hierarchy, you group the rows of the flat data before passing the data to the AdvancedDataGrid control. The following code contains a variation on the hierarchical data shown in the previous image, but arranges the data in a flat data structure:

```
[Bindable]
private var dpFlat:ArrayCollection = new ArrayCollection([
  {Region:"Southwest", Territory:"Arizona",
      Territory_Rep:"Barbara Jennings", Actual:38865, Estimate:40000},
```

```
  {Region:"Southwest", Territory:"Arizona",
      Territory_Rep:"Dana Binn", Actual:29885, Estimate:30000},
  {Region:"Southwest", Territory:"Central California",
      Territory_Rep:"Joe Smith", Actual:29134, Estimate:30000},
  {Region:"Southwest", Territory:"Nevada",
      Territory_Rep:"Bethany Pittman", Actual:52888, Estimate:45000},
  {Region:"Southwest", Territory:"Northern California",
      Territory_Rep:"Lauren Ipsum", Actual:38805, Estimate:40000},
  {Region:"Southwest", Territory:"Northern California",
      Territory_Rep:"T.R. Smith", Actual:55498, Estimate:40000},
  {Region:"Southwest", Territory:"Southern California",
      Territory_Rep:"Alice Treu", Actual:44985, Estimate:45000},
  {Region:"Southwest", Territory:"Southern California",
      Territory_Rep:"Jane Grove", Actual:44913, Estimate:45000}
]);
```

In this example, the data contains a single level of individual records with no inherent hierarchy. To group the data, you specify one or more data fields used to arrange the data in a hierarchy. The following example shows the AdvancedDataGrid control where the flat data has been grouped by the Region field of the data:

The code for this example is in the section "Displaying grouped data" on page 296.

Setting the data provider with hierarchical data

To configure the AdvancedDataGrid control to display hierarchical data and the navigation tree, you pass to the `dataProvider` property an instance of the HierarchicalData class or of the GroupingCollection class. Use the HierarchicalData class when your data is already arranged in a hierarchy. Use the GroupingCollection class when your data is in a flat structure. As part of configuring an instance of the GroupingCollection class, you specify one or more data fields that are used to arrange the flat data in a hierarchy.

For more information on displaying hierarchical data, see "Displaying hierarchical data" on page 294. For more information on displaying grouped data, see "Displaying grouped data" on page 296.

You can create an instance of the HierarchicalData class or an instance of the GroupingCollection class from any data that you can use as a data provider. However, the AdvancedDataGrid control modifies its internal representation of the data as follows:

- An Array is represented internally by the AdvancedDataGrid control as an instance of the ArrayCollection class.
- An ArrayCollection class is represented internally by the AdvancedDataGrid control as an instance of the ArrayCollection class.
- A String that contains valid XML text is represented internally by the AdvancedDataGrid control as an instance of the XMLListCollection class.
- An XMLNode class or an XMLList class is represented internally by the AdvancedDataGrid control as an instance of the XMLListCollection class.

- Any object that implements the ICollectionView interface is represented internally by the AdvancedDataGrid control as an instance of the ICollectionView interface.
- An object of any other data type is wrapped in an Array instance with the object as its sole entry.

For example, you use an Array to create an instance of the HierarchicalData class, and then pass that HierarchicalData instance to the `AdvancedDataGrid.dataProvider` property. If you read the data back from the `AdvancedDataGrid.dataProvider` property, it is returned as an ArrayCollection instance.

Calling validateNow() after setting the data provider

In some situations, you may set the data provider of the AdvancedDataGrid control to hierarchical or grouped data, and then immediately try to perform an action based on the new data provider. This typically occurs when you set the `dataProvider` property in ActionScript, as the following example shows:

```
adg.dataProvider = groupedCollection;
adg.expandAll();
```

In this example, the call to `expandAll()` fails because the AdvancedDataGrid control is in the process of setting the `dataProvider` property, and the `expandAll()` method either processes the old value of the `dataProvider` property, if one existed, or does nothing.

In this situation, you must insert the `validateNow()` method after setting the data provider. The `validateNow()` method validates and updates the properties and layout of the control, and then redraws it, if necessary. The following example inserts the `validateNow()` method before the `expandAll()` method:

```
adg.dataProvider = groupedCollection;
adg.validateNow();
adg.expandAll();
```

Do not insert the `validateNow()` method every time you set the `dataProvider` property because it can affect the performance of your application; it is only required in some situations when you attempt to perform an operation on the control immediately after setting the `dataProvider` property.

Controlling the navigation tree of the AdvancedDataGrid control

The AdvancedDataGrid control is sometimes referred to as a *tree datagrid* because a column of the control uses an expandable tree to determine which rows are currently visible in the control. Often, the tree appears in the left-most column of the control, where the first column of the control is associated with a field of the control's data provider. That data field of the data provider provides the text for the labels of the tree nodes.

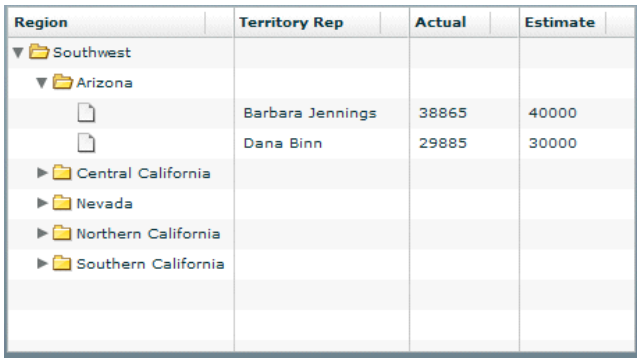
In the following example, you populate the AdvancedDataGrid control with the hierarchal data structure shown in "Hierarchical and grouped data display" on page 287. The data contains a top-level object that contains a Region field, and multiple second-level children. Each second-level child also contains a Region field and one or more children.

The AdvancedDataGrid control in this example defines four columns to display the data: Region, Territory Rep, Actual, and Estimate.

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

  <mx:Script>
    <![CDATA[
          import mx.collections.ArrayCollection;

          include "SimpleHierarchicalData.as";
      ]]>
    </mx:Script>
```

```
    <mx:AdvancedDataGrid width="100%" height="100%">
        <mx:dataProvider>
            <mx:HierarchicalData source="{dpHierarchy}"/>
        </mx:dataProvider>
        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumn dataField="Actual"/>
            <mx:AdvancedDataGridColumn dataField="Estimate"/>
        </mx:columns>
    </mx:AdvancedDataGrid>
</mx:Application>
```

The following image shows the AdvancedDataGrid control created by this example. The control displays a folder icon to represent branch nodes of the tree, and a file icon to represent leaf nodes. The first column of the control is associated with the Region field of the data provider, so the tree labels display the value of the Region field.



Notice that the leaf icon for the tree does not show a label. This is because the individual records for each territory representative do not contain a Region field.

While the tree is often positioned in the left-most column of the control, you can use the `treeColumn` property to specify any column in the control, as the following example shows:

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

  <mx:Script>
    <![CDATA[
          import mx.collections.ArrayCollection;

          include "HierarchicalDataCategories.as";
      ]]>
    </mx:Script>

    <mx:AdvancedDataGrid
        width="100%" height="100%"
        treeColumn="{rep}">
        <mx:dataProvider>
            <mx:HierarchicalData source="{dpHierarchy}"
                childrenField="categories"/>
        </mx:dataProvider>
        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn id="rep"
```

```
                dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumn dataField="Actual"/>
            <mx:AdvancedDataGridColumn dataField="Estimate"/>
        </mx:columns>
    </mx:AdvancedDataGrid>
</mx:Application>
```

Setting navigation tree icons and labels

The navigation tree lets you control the icons and labels used for the branch and leaf nodes. For example, you can display a tree of labels with no icons, a tree with just folder icons, a tree with no labels at all, or a tree in its own column that is not associated with any data field.

The following table describes the style properties of the AdvancedDataGrid control that you use to set the tree icons:

Style property	Description
`defaultLeafIcon`	Specifies the leaf icon.
`disclosureClosedIcon`	Specifies the icon that is displayed next to a closed branch node. The default icon is a black triangle.
`disclosureOpenIcon`	Specifies the icon that is displayed next to an open branch node. The default icon is a black triangle.
`folderClosedIcon`	Specifies the folder closed icon for a branch node.
`folderOpenIcon`	Specifies the folder open icon for a branch node.

The following example sets the default leaf icon to null to hide it, and uses custom icons for the folder open and closed icons:

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

  <mx:Script>
    <![CDATA[
          import mx.collections.ArrayCollection;

          include "SimpleHierarchicalData.as";
      ]]>
    </mx:Script>

    <mx:AdvancedDataGrid width="100%" height="100%"
        defaultLeafIcon="{null}"
        folderOpenIcon="@Embed(source='assets/folderOpenIcon.jpg')"
        folderClosedIcon="@Embed(source='assets/folderClosedIcon.jpg')">
        <mx:dataProvider>
            <mx:HierarchicalData source="{dpHierarchy}"/>
        </mx:dataProvider>
        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumn dataField="Actual"/>
            <mx:AdvancedDataGridColumn dataField="Estimate"/>
        </mx:columns>
    </mx:AdvancedDataGrid>
</mx:Application>
```

You can also specify the styles by using the `setStyle()` method, and by using an `<mx:Style>` tag, as the following example shows:

```
<mx:Style>
    AdvancedDataGrid {
        defaultLeafIcon:ClassReference(null);
        folderOpenIcon:Embed(source='assets/folderOpenIcon.jpg');
        folderClosedIcon:Embed(source='assets/folderClosedIcon.jpg');
    }
</mx:Style>
```

Using the groupIconFunction and groupLabelFunction properties

Use the `groupIconFunction` and `groupLabelFunction` properties of the AdvancedDataGrid class to define callback functions to control the display of the leaf nodes of the navigation tree. The callback function specified by the `groupIconFunction` property controls the icon, and the callback function specified by the `groupLabelFunction` controls the label.

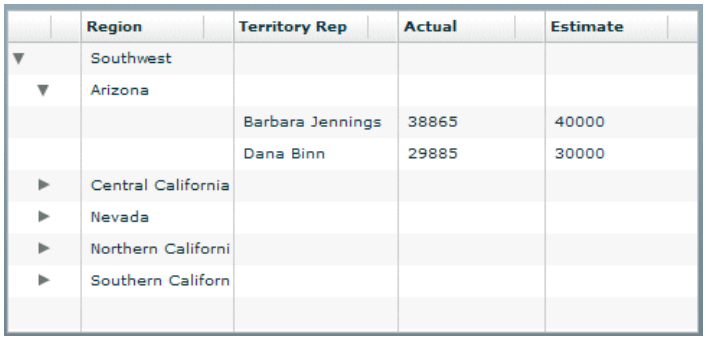
The following example uses the `groupIconFunction` property to display a custom icon for the top level node of the navigation tree, and display the default icon for all other leaf nodes of the tree:

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

  <mx:Script>
    <![CDATA[
          import mx.collections.ArrayCollection;

          include "SimpleHierarchicalData.as";

         // Embed the icon for the groups.
         [Bindable]
         [Embed(source="assets/topTreeIcon.png")]
         public var icn:Class;

         // Define the groupIconFunction callback function.
         public function myIconFunc(item:Object, depth:int):Class {
             if(depth == 1)
                 // If this is the top-level of the tree, return the icon.
                 return icn;
             else
                 // If this is any other level, return null.
                 return null;
         }
      ]]>
    </mx:Script>

    <mx:AdvancedDataGrid
        width="100%" height="100%"
        groupIconFunction="myIconFunc">
        <mx:dataProvider>
            <mx:HierarchicalData source="{dpHierarchy}"/>
        </mx:dataProvider>
        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumn dataField="Actual"/>
            <mx:AdvancedDataGridColumn dataField="Estimate"/>
        </mx:columns>
    </mx:AdvancedDataGrid>
</mx:Application>
```

Creating a separate column for the navigation tree

In the examples in "Hierarchical and grouped data display" on page 287, the first column also displays the Region field of the data. Therefore, as you expand the nodes of the navigation tree, the label of each tree node corresponds to the value of the Region field in the data provider for that row. For the leaf nodes of the tree, the data provider does not contain a value for the Region field, so the labels for the leaf nodes are blank.

You can also put the navigation tree in its own column, where the column is not associated with a data field, as the following example shows:



This example does not associate a data field with the column that contains the tree, so the tree icons appear with no label. This example also sets the `folderClosedIcon`, `folderOpenIcon`, and `defaultLeafIcon` properties of the AdvancedDataGrid control to null, but does display the disclosure icons so that the user can still open and close tree nodes.

The following code implements this example. Notice that the first column is not associated with a data field. Because it is the first column in the data grid, the AdvancedDataGrid control automatically uses it to display the navigation tree.

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

  <mx:Script>
    <![CDATA[
          import mx.collections.ArrayCollection;

          include "SimpleHierarchicalData.as";
      ]]>
    </mx:Script>

    <mx:AdvancedDataGrid width="100%" height="100%"
        folderClosedIcon="{null}"
        folderOpenIcon="{null}"
        defaultLeafIcon="{null}">
        <mx:dataProvider>
            <mx:HierarchicalData source="{dpHierarchy}"/>
        </mx:dataProvider>
        <mx:columns>
            <mx:AdvancedDataGridColumn headerText="" width="50"/>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumn dataField="Actual"/>
            <mx:AdvancedDataGridColumn dataField="Estimate"/>
        </mx:columns>
```

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

Displaying hierarchical data

Hierarchical data is data in a structured format where the data is already arranged in a hierarchy. To display hierarchical data in the AdvancedDataGrid control, you set the data provider of the control to an instance of the HierarchicalData class. The structure of the data in the data provider defines how the AdvancedDataGrid control displays the data.

Defining hierarchical data with an ArrayCollection

Using an ArrayCollection is a common way to create hierarchical data, as the following example shows in the SimpleHierarchicalData.as file. In this example, the data has three levels – a root level and two child levels:

```
[Bindable]
private var dpHierarchy:ArrayCollection = new ArrayCollection([
  {Region:"Southwest", children: [
     {Region:"Arizona", children: [
        {Territory_Rep:"Barbara Jennings", Actual:38865, Estimate:40000},
        {Territory_Rep:"Dana Binn", Actual:29885, Estimate:30000}]},
     {Region:"Central California", children: [
        {Territory_Rep:"Joe Smith", Actual:29134, Estimate:30000}]},
     {Region:"Nevada", children: [
        {Territory_Rep:"Bethany Pittman", Actual:52888, Estimate:45000}]},
     {Region:"Northern California", children: [
        {Territory_Rep:"Lauren Ipsum", Actual:38805, Estimate:40000},
        {Territory_Rep:"T.R. Smith", Actual:55498, Estimate:40000}]},
     {Region:"Southern California", children: [
        {Territory_Rep:"Alice Treu", Actual:44985, Estimate:45000},
        {Territory_Rep:"Jane Grove", Actual:44913, Estimate:45000}]}
  ]}
]);
```

This example uses the `children` keyword in the ArrayCollection definition to define the data hierarchy. The `children` keyword is the default keyword used by the HierarchicalData class to define the hierarchy.

You can use a different keyword to define the hierarchy. The following example shows the HierarchicalDataCategories.as file, which uses the `categories` keyword:

```
[Bindable]
private var dpHierarchy:ArrayCollection = new ArrayCollection([
  {Region:"Southwest", categories: [
     {Region:"Arizona", categories: [
        {Territory_Rep:"Barbara Jennings", Actual:38865, Estimate:40000},
        {Territory_Rep:"Dana Binn", Actual:29885, Estimate:30000}]},
     {Region:"Central California", categories: [
        {Territory_Rep:"Joe Smith", Actual:29134, Estimate:30000}]},
     {Region:"Nevada", categories: [
        {Territory_Rep:"Bethany Pittman", Actual:52888, Estimate:45000}]},
     {Region:"Northern California", categories: [
        {Territory_Rep:"Lauren Ipsum", Actual:38805, Estimate:40000},
        {Territory_Rep:"T.R. Smith", Actual:55498, Estimate:40000}]},
     {Region:"Southern California", categories: [
        {Territory_Rep:"Alice Treu", Actual:44985, Estimate:45000},
        {Territory_Rep:"Jane Grove", Actual:44913, Estimate:45000}]}
  ]}
]);
```

Use the `HierarchicalData.childrenField` property to specify the name of the field that defines the hierarchy, as the following example shows:

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

  <mx:Script>
    <![CDATA[
          import mx.collections.ArrayCollection;

          include "HierarchicalDataCategories.as";
      ]]>
    </mx:Script>

    <mx:AdvancedDataGrid width="100%" height="100%">
        <mx:dataProvider>
            <mx:HierarchicalData source="{dpHierarchy}"
                childrenField="categories"/>
        </mx:dataProvider>
        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumn dataField="Actual"/>
            <mx:AdvancedDataGridColumn dataField="Estimate"/>
        </mx:columns>
    </mx:AdvancedDataGrid>
</mx:Application>
```

Displaying hierarchical XML data

The examples in the previous section use an ArrayCollection to populate the AdvancedDataGrid control. You can also populate the control with hierarchical XML data. The following example modifies the data from the previous section to format it as XML, and then passes that data to the AdvancedDataGrid control:

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

    <mx:Script>
        <![CDATA[
            import mx.collections.HierarchicalData;
            import mx.collections.XMLListCollection;
        ]]>
    </mx:Script>

    <mx:XMLList id="dpHierarchyXML" >
        <Region Region="Southwest">
            <Region Region="Arizona">
                <Territory_Rep Territory_Rep="Barbara Jennings"
                    Actual="38865" Estimate="40000"/>
                <Territory_Rep Territory_Rep="Dana Binn"
                    Actual="29885" Estimate="30000"/>
            </Region>
            <Region Region="Central California">
                <Territory_Rep Territory_Rep="Joe Smith"
                    Actual="29134" Estimate="30000"/>
            </Region>
            <Region Region="Nevada">
                <Territory_Rep Territory_Rep="Bethany Pittman"
```

```
                Actual="52888" Estimate="45000"/>
            </Region>
            <Region Region="Northern California">
                <Territory_Rep Territory_Rep="Lauren Ipsum"
                    Actual="38805" Estimate="40000"/>
                <Territory_Rep Territory_Rep="T.R. Smith"
                    Actual="55498" Estimate="40000"/>
            </Region>
            <Region Region="Southern California">
                <Territory_Rep Territory_Rep="Alice Treu"
                    Actual="44985" Estimate="45000"/>
                <Territory_Rep Territory_Rep="Jane Grove"
                    Actual="44913" Estimate="45000"/>
            </Region>
        </Region>
    </mx:XMLList>

    <mx:AdvancedDataGrid width="100%" height="100%"
        dataProvider="{new HierarchicalData(dpHierarchyXML)}">
        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="@Region"
                headerText="Region"/>
            <mx:AdvancedDataGridColumn dataField="@Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumn dataField="@Actual"
                headerText="Actual"/>
            <mx:AdvancedDataGridColumn dataField="@Estimate"
                headerText="Estimate"/>
        </mx:columns>
    </mx:AdvancedDataGrid>
</mx:Application>
```

Displaying grouped data

Grouped data is flat data that you arrange in a hierarchy for display in the AdvancedDataGrid control. To group your data, you specify one or more data fields that collect the data into the hierarchy.

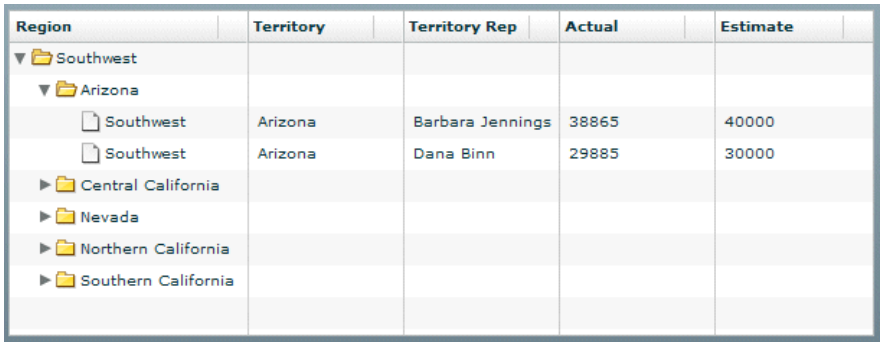
To populate the AdvancedDataGrid control with grouped data, create an instance of the GroupingCollection class from your flat data, and then pass that GroupingCollection instance to the data provider of the AdvancedDataGrid control. When you create the instance of the GroupingCollection from your flat data, you specify the fields for the data that are used to create the hierarchy.

Most of the examples in this section use the following flat data to create an instance of the GroupingCollection class:

```
[Bindable]
private var dpFlat:ArrayCollection = new ArrayCollection([
  {Region:"Southwest", Territory:"Arizona",
      Territory_Rep:"Barbara Jennings", Actual:38865, Estimate:40000},
  {Region:"Southwest", Territory:"Arizona",
      Territory_Rep:"Dana Binn", Actual:29885, Estimate:30000},
  {Region:"Southwest", Territory:"Central California",
      Territory_Rep:"Joe Smith", Actual:29134, Estimate:30000},
  {Region:"Southwest", Territory:"Nevada",
      Territory_Rep:"Bethany Pittman", Actual:52888, Estimate:45000},
  {Region:"Southwest", Territory:"Northern California",
      Territory_Rep:"Lauren Ipsum", Actual:38805, Estimate:40000},
  {Region:"Southwest", Territory:"Northern California",
      Territory_Rep:"T.R. Smith", Actual:55498, Estimate:40000},
```

```
  {Region:"Southwest", Territory:"Southern California",
      Territory_Rep:"Alice Treu", Actual:44985, Estimate:45000},
  {Region:"Southwest", Territory:"Southern California",
      Territory_Rep:"Jane Grove", Actual:44913, Estimate:45000}
]);
```

The following image shows an AdvancedDataGrid control that uses this data as input. This example specifies two fields that are used to group the data: Region and Territory. The first column of the AdvancedDataGrid control is associated with the Region field, so the navigation tree uses the Region field to define the labels for the leaf nodes of the tree.



The following code implements this example:

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

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            include "SimpleFlatData.as"
        ]]>
    </mx:Script>

    <mx:AdvancedDataGrid id="myADG"
        width="100%" height="100%"
        initialize="gc.refresh();">
        <mx:dataProvider>
            <mx:GroupingCollection id="gc" source="{dpFlat}">
                <mx:grouping>
                    <mx:Grouping>
                        <mx:GroupingField name="Region"/>
                        <mx:GroupingField name="Territory"/>
                    </mx:Grouping>
                </mx:grouping>
            </mx:GroupingCollection>
        </mx:dataProvider>

        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumn dataField="Actual"/>
            <mx:AdvancedDataGridColumn dataField="Estimate"/>
        </mx:columns>
```

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

The GroupingCollection instance reformats the data based on these fields so that internally it is represented by the GroupingCollection instance, as the following example shows:

```
[{GroupLabel:"Southwest", children:[
    {GroupLabel:"Arizona", children:[
      {Region:"Southwest", Territory:"Arizona",
        Territory_Rep:"Barbara Jennings", Actual:38865, Estimate:40000},
      {Region:"Southwest", Territory:"Arizona",
        Territory_Rep:"Dana Binn", Actual:29885, Estimate:30000}]}
    {GroupLabel:"Central California", children:[
      {Region:"Southwest", Territory:"Central California",
        Territory_Rep:"Joe Smith", Actual:29134, Estimate:30000}]}
    {GroupLabel:"Nevada", children:[
      {Region:"Southwest", Territory:"Nevada",
        Territory_Rep:"Bethany Pittman", Actual:52888, Estimate:45000}]}
    {GroupLabel:"Northern California", children:[
      {Region:"Southwest", Territory:"Northern California",
        Territory_Rep:"Lauren Ipsum", Actual:38805, Estimate:40000},
      {Region:"Southwest", Territory:"Northern California",
        Territory_Rep:"T.R. Smith", Actual:55498, Estimate:40000}]}
    {GroupLabel:"Southern California", children:[
      {Region:"Southwest", Territory:"Southern California",
        Territory_Rep:"Alice Treu", Actual:44985, Estimate:45000},
      {Region:"Southwest", Territory:"Southern California",
        Territory_Rep:"Jane Grove", Actual:44913, Estimate:45000}]}
]}]
```

Notice that the representation consists of a data hierarchy based on the Region and Territory fields of the flat data. The GroupingCollection instance still contains the original rows of the input flat data, but those rows are now arranged in a hierarchy based on the grouping fields.

Calling the GroupingCollection.refresh() method

The `GroupingCollection.refresh()` method applies the settings of the GroupingCollection class to the data. You must call this method any time you modify the GroupingCollection class, such as by setting the `grouping`, `source`, or `summaries` properties of the GroupingCollection class. You also call the `GroupingCollection.refresh()` method when you modify a GroupingField of the GroupingCollection class, such as by changing the `caseInsensitive`, `compareFunction`, or `groupingFunction` properties.

Notice that in the previous example, the AdvancedDataGrid control calls the `GroupingCollection.refresh()` method in response to the `initialize` event of the AdvancedDataGrid control.

Using the default MXML property of the GroupingCollection class

The `grouping` property is the default MXML property of the GroupingCollection class, so you can rewrite the previous example as follows:

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

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            include "SimpleFlatData.as"
        ]]>
```

```
    </mx:Script>

    <mx:AdvancedDataGrid id="myADG"
        width="100%" height="100%"
        initialize="gc.refresh();">
        <mx:dataProvider>
            <mx:GroupingCollection id="gc" source="{dpFlat}">
                    <mx:Grouping>
                        <mx:GroupingField name="Region"/>
                        <mx:GroupingField name="Territory"/>
                    </mx:Grouping>
            </mx:GroupingCollection>
        </mx:dataProvider>

        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumn dataField="Actual"/>
            <mx:AdvancedDataGridColumn dataField="Estimate"/>
        </mx:columns>
    </mx:AdvancedDataGrid>
</mx:Application>
```

Setting the name of the GroupLabel field

By default, GroupLabel is the field name for the data items added to the flat data to create the hierarchy. For an example of the data structure created by the GroupingCollection class that uses the default field name, see "Displaying grouped data" on page 296.

Use the `Grouping.label` property to specify a different name so that you can create a different group label for each level of the generated hierarchy.

Creating a column for the GroupLabel field

The AdvancedDataGrid control uses the GroupLabel field to define the labels for the branch nodes of the navigation tree. One option when displaying grouped data is to create a column for the top-level data items that the grouping fields create. For example, you group your flat data by the Region and Territory fields. These fields appear as the labels for the branch nodes of the navigation tree, so you omit separate columns for those fields, as the following example shows:

The following code creates this example. Notice that this example includes an AdvancedDataGridColumn instance for the GroupLabel field, and no column definitions for the Region and Territory fields.

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

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            include "SimpleFlatData.as";
        ]]>
    </mx:Script>

    <mx:AdvancedDataGrid id="myADG"
        width="100%" height="100%"
        defaultLeafIcon="{null}"
        initialize="gc.refresh();">
        <mx:dataProvider>
            <mx:GroupingCollection id="gc" source="{dpFlat}">
                <mx:Grouping>
                    <mx:GroupingField name="Region"/>
                    <mx:GroupingField name="Territory"/>
                </mx:Grouping>
            </mx:GroupingCollection>
        </mx:dataProvider>

        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="GroupLabel"
                headerText="Region/Territory"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumn dataField="Actual"/>
            <mx:AdvancedDataGridColumn dataField="Estimate"/>
        </mx:columns>
   </mx:AdvancedDataGrid>
</mx:Application>
```

Creating groups in ActionScript

The example in the previous section created the groups in MXML. However, you could let the user define the grouping at run time. The following example creates the groups in ActionScript by using an event handler:

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

    <mx:Script>
      <![CDATA[
        import mx.collections.GroupingField;
        import mx.collections.Grouping;
        import mx.collections.GroupingCollection;
        import mx.collections.ArrayCollection;

        include "SimpleFlatData.as";

        [Bindable]
        public var myGColl:GroupingCollection = new GroupingCollection();

        private var myGrp:Grouping = new Grouping();
```

```
        private function initDG():void {

            // Initialize the GroupingCollection instance.
            myGColl.source = dpFlat;

            // The Grouping instance defines the grouping fields
            // in the collection, and the order of the groups
            // in the hierarchy.
            myGrp.fields =
              [new GroupingField("Region"), new GroupingField("Territory")];

            // The grouping property contains a Grouping instance.
            myGColl.grouping = myGrp;

            // Specify the GroupedCollection as the data provider for
            // the AdvancedDataGrid control.
            myADG.dataProvider=myGColl;

            // Refresh the display.
            myGColl.refresh();
        }
      ]]>
    </mx:Script>

    <mx:AdvancedDataGrid id="myADG"
        width="100%" height="100%"
        creationComplete="initDG();">
        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumn dataField="Actual"/>
            <mx:AdvancedDataGridColumn dataField="Estimate"/>
        </mx:columns>
   </mx:AdvancedDataGrid>
</mx:Application>
```

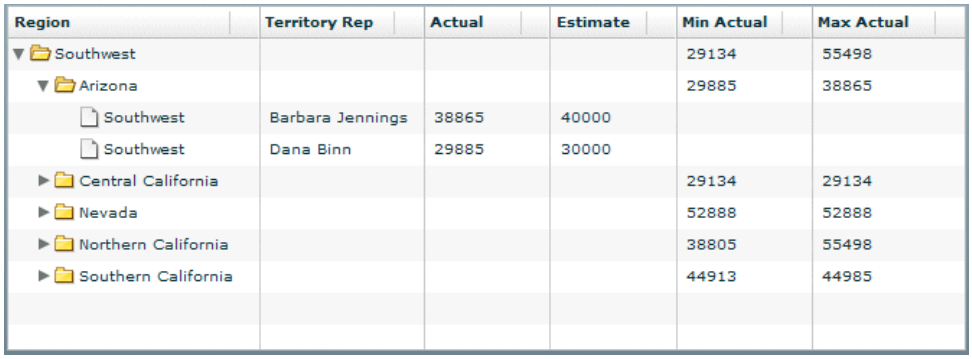
Creating summary rows

Summary data provides a way for you to extract information from your data for display in the AdvancedDataGrid control. For example, you can add summary data that contains the average value of a field across all rows of your data, that contains the maximum field value, the minimum field value, or that contains other types of summaries.

Note: *Summary data is not supported for hierarchical data represented by the HierarchicalData class. You can only create summary data for data represented by the GroupingCollection class.*

You create summary data about your groups by using the `summaries` property of the GroupingField class. You can display the summary data in an existing row of the AdvancedDataGrid control, or display it in a separate row.

The following example shows an AdvancedDataGrid control that displays two summary fields, Min Actual and Max Actual:



Region	Territory Rep	Actual	Estimate	Min Actual	Max Actual
Southwest				29134	55498
Arizona				29885	38865
Southwest	Barbara Jennings	38865	40000		
Southwest	Dana Binn	29885	30000		
Central California				29134	29134
Nevada				52888	52888
Northern California				38805	55498
Southern California				44913	44985

The Min Actual and Max Actual fields in the top row correspond to summaries for all rows in the group, and the Min Actual and Max Actual fields for each Territory correspond to summaries for all rows in the territory subgroup.

The following code creates this control:

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

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            include "SimpleFlatData.as"
      ]]>
    </mx:Script>

    <mx:AdvancedDataGrid id="myADG"
        width="100%" height="100%"
        initialize="gc.refresh();">
        <mx:dataProvider>
            <mx:GroupingCollection id="gc" source="{dpFlat}">
                <mx:Grouping>
                    <mx:GroupingField name="Region">
                        <mx:summaries>
                          <mx:SummaryRow summaryPlacement="group">
                            <mx:fields>
                                <mx:SummaryField dataField="Actual"
                                    label="Min Actual" operation="MIN"/>
                                <mx:SummaryField dataField="Actual"
                                    label="Max Actual" operation="MAX"/>
                            </mx:fields>
                          </mx:SummaryRow>
                        </mx:summaries>
                    </mx:GroupingField>
                    <mx:GroupingField name="Territory">
                        <mx:summaries>
                          <mx:SummaryRow summaryPlacement="group">
                            <mx:fields>
                                <mx:SummaryField dataField="Actual"
                                    label="Min Actual" operation="MIN"/>
                                <mx:SummaryField dataField="Actual"
                                    label="Max Actual" operation="MAX"/>
```

```
                        </mx:fields>
                    </mx:SummaryRow>
                </mx:summaries>
            </mx:GroupingField>
        </mx:Grouping>
    </mx:GroupingCollection>
</mx:dataProvider>

<mx:columns>
    <mx:AdvancedDataGridColumn dataField="Region"/>
    <mx:AdvancedDataGridColumn dataField="Territory_Rep"
        headerText="Territory Rep"/>
    <mx:AdvancedDataGridColumn dataField="Actual"/>
    <mx:AdvancedDataGridColumn dataField="Estimate"/>
    <mx:AdvancedDataGridColumn dataField="Min Actual"/>
    <mx:AdvancedDataGridColumn dataField="Max Actual"/>
</mx:columns>
</mx:AdvancedDataGrid>
</mx:Application>
```

Notice in this example that you use the `GroupingField.summaries` property to specify the SummaryRow instance. The SummaryRow instance contains all the information about a data summary. In this example, you use the `summaryPlacement` property to add the summary data to the grouped data. Optionally, you can add new rows to contain the summary data. For more information, see "Specifying the display location of the summary data in the AdvancedDataGrid control" on page 305.

Each SummaryRow instance specifies one or more SummaryField instances that create the data summary. In this example, you use the `dataField` property to specify that the summaries are determined by the data in the Actual data field, the `label` property to specify the name of the data field created to hold the summary data, and the `operation` property to specify how to create the summary for numeric fields. You can specify one of the following values: `SUM`, `MIN`, `MAX`, `AVG`, or `COUNT`.

Internally, the GroupingCollection represents the grouped data, with the new summary data fields, as the following example shows:

```
[{GroupLabel:"Southwest", Max Actual:55498, Min Actual:29134, children:[
  {GroupLabel:"Arizona", Max Actual:38865, Min Actual:29885, children:[
    {Region:"Southwest", Territory:"Arizona",
      Territory_Rep:"Barbara Jennings", Actual:38865, Estimate:40000},
    {Region:"Southwest", Territory:"Arizona",
      Territory_Rep:"Dana Binn", Actual:29885, Estimate:30000}]}
  {GroupLabel:"Central California", Max Actual:29134, Min Actual:29134, children:[
    {Region:"Southwest", Territory:"Central California",
      Territory_Rep:"Joe Smith", Actual:29134, Estimate:30000}]}
  {GroupLabel:"Nevada", Max Actual:52888, Min Actual:52888, children:[
    {Region:"Southwest", Territory:"Nevada",
      Territory_Rep:"Bethany Pittman", Actual:52888, Estimate:45000}]}
  {GroupLabel:"Northern California", Max Actual:55498, Min Actual:38805, children:[
    {Region:"Southwest", Territory:"Northern California",
      Territory_Rep:"Lauren Ipsum", Actual:38805, Estimate:40000},
    {Region:"Southwest", Territory:"Northern California",
      Territory_Rep:"T.R. Smith", Actual:55498, Estimate:40000}]}
  {GroupLabel:"Southern California", Max Actual:44985, Min Actual:44913, children:[
    {Region:"Southwest", Territory:"Southern California",
      Territory_Rep:"Alice Treu", Actual:44985, Estimate:45000},
    {Region:"Southwest", Territory:"Southern California",
      Territory_Rep:"Jane Grove", Actual:44913, Estimate:45000}]}
]}]
```

Using the default properties of the GroupingField and SummaryRow classes

The `GroupingField.summaries` property is the default property for the GroupingField class, and the `SummaryRow.fields` property is the default property for the SummaryRow class; therefore, you can omit those properties in your code and rewrite the previous example, as the following example shows:

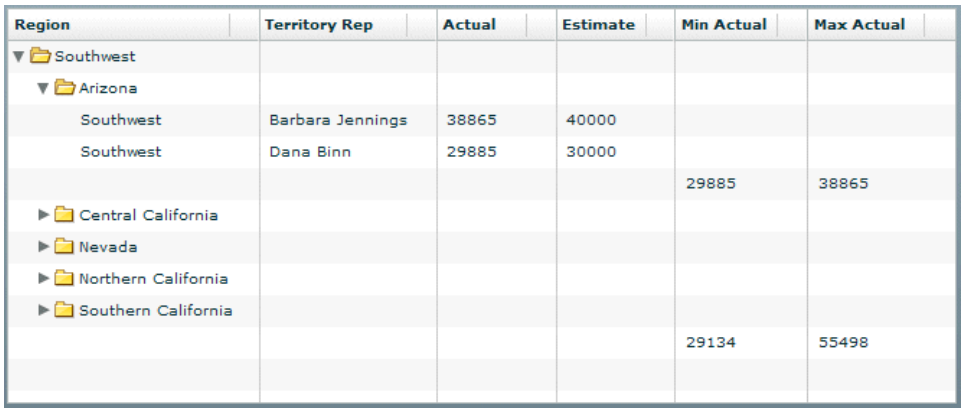
```
<?xml version="1.0"?>
<!-- dpcontrols/adg/SummaryGroupADGDefProp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            include "SimpleFlatData.as"
      ]]>
    </mx:Script>

    <mx:AdvancedDataGrid id="myADG"
        width="100%" height="100%"
        initialize="gc.refresh();">
        <mx:dataProvider>
            <mx:GroupingCollection id="gc" source="{dpFlat}">
                <mx:Grouping>
                    <mx:GroupingField name="Region">
                          <mx:SummaryRow summaryPlacement="group">
                                <mx:SummaryField operation="MIN"
                                    dataField="Actual" label="Min Actual"/>
                                <mx:SummaryField operation="MAX"
                                    dataField="Actual" label="Max Actual"/>
                          </mx:SummaryRow>
                    </mx:GroupingField>
                    <mx:GroupingField name="Territory">
                          <mx:SummaryRow summaryPlacement="group">
                                <mx:SummaryField operation="MIN"
                                    dataField="Actual" label="Min Actual"/>
                                <mx:SummaryField operation="MAX"
                                    dataField="Actual" label="Max Actual"/>
                          </mx:SummaryRow>
                    </mx:GroupingField>
                </mx:Grouping>
            </mx:GroupingCollection>
        </mx:dataProvider>

        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumn dataField="Actual"/>
            <mx:AdvancedDataGridColumn dataField="Estimate"/>
            <mx:AdvancedDataGridColumn dataField="Min Actual"/>
            <mx:AdvancedDataGridColumn dataField="Max Actual"/>
        </mx:columns>
   </mx:AdvancedDataGrid>
</mx:Application>
```

Specifying the display location of the summary data in the AdvancedDataGrid control

The SummaryRow class contains the `summaryPlacement` property that determines where the summary data appears in the AdvancedDataGrid control. The following table describes the values of the `summaryPlacement` property:

Value of summaryPlacement property	Description
`first`	Creates a summary row as the first row in the group.
`last`	Creates a summary row as the last row in the group.
`group`	Adds the summary data to the row corresponding to the group.

The section "Creating summary rows" on page 301 shows an example of summary data added to the group by specifying `group` as the value of the `summaryPlacement` property. The following example shows the same grouped data but with the `summaryPlacement` property set to `last`:



Region	Territory Rep	Actual	Estimate	Min Actual	Max Actual
▼ Southwest					
▼ Arizona					
Southwest	Barbara Jennings	38865	40000		
Southwest	Dana Binn	29885	30000		
				29885	38865
▶ Central California					
▶ Nevada					
▶ Northern California					
▶ Southern California					
				29134	55498

You can specify multiple values to the `summaryPlacement` property, separated by a space; for example, a value of `last group` specifies to show the summary at the group level and as the last row of the group.

Internally, the GroupingCollection represents the grouped data, with the new summary data fields, as the following example shows:

```
[{GroupLabel:"Southwest", children:[
  {GroupLabel:"Arizona", children:[
    {Region:"Southwest", Territory:"Arizona",
      Territory_Rep:"Barbara Jennings", Actual:38865, Estimate:40000},
    {Region:"Southwest", Territory:"Arizona",
      Territory_Rep:"Dana Binn", Actual:29885, Estimate:30000},
    {Max Actual:38865, Min Actual:29885}]}
  {GroupLabel:"Central California", children:[
    {Region:"Southwest", Territory:"Central California",
      Territory_Rep:"Joe Smith", Actual:29134, Estimate:30000},
    {Max Actual:29134, Min Actual:29134}]}
  {GroupLabel:"Nevada", children:[
    {Region:"Southwest", Territory:"Nevada",
      Territory_Rep:"Bethany Pittman", Actual:52888, Estimate:45000},
    {Max Actual:52888, Min Actual:52888}]}
  {GroupLabel:"Northern California", children:[
    {Region:"Southwest", Territory:"Northern California",
      Territory_Rep:"Lauren Ipsum", Actual:38805, Estimate:40000},
    {Region:"Southwest", Territory:"Northern California",
```

```
      Territory_Rep:"T.R. Smith", Actual:55498, Estimate:40000},
    {Max Actual:55498, Min Actual:38805}]}
  {GroupLabel:"Southern California", children:[
    {Region:"Southwest", Territory:"Southern California",
      Territory_Rep:"Alice Treu", Actual:44985, Estimate:45000},
    {Region:"Southwest", Territory:"Southern California",
      Territory_Rep:"Jane Grove", Actual:44913, Estimate:45000},
    {Max Actual:44985, Min Actual:44913}]}
  {Max Actual:55498, Min Actual:29134}
]}]
```

Notice that a new row is added to the entire group to hold the summary data, and a new row is added to each subgroup to hold its summary data.

Creating multiple summaries

You can specify multiple SummaryRow instances in a single GroupingField instance. In the following example, you create summary data to define the following fields: Min Actual, Max Actual, Min Estimate, and Max Estimate for the Region group:

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

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            include "SimpleFlatData.as"
      ]]>
    </mx:Script>

    <mx:AdvancedDataGrid id="myADG"
        width="100%" height="100%"
        initialize="gc.refresh();">
        <mx:dataProvider>
            <mx:GroupingCollection id="gc" source="{dpFlat}">
                <mx:Grouping>
                    <mx:GroupingField name="Region">
                          <mx:SummaryRow summaryPlacement="group">
                                <mx:SummaryField operation="MIN"
                                   dataField="Actual" label="Min Actual"/>
                                <mx:SummaryField operation="MAX"
                                   dataField="Actual" label="Max Actual"/>
                          </mx:SummaryRow>
                          <mx:SummaryRow summaryPlacement="group">
                                <mx:SummaryField operation="MIN"
                                   dataField="Estimate" label="Min Estimate"/>
                                <mx:SummaryField operation="MAX"
                                   dataField="Estimate" label="Max Estimate"/>
                          </mx:SummaryRow>
                    </mx:GroupingField>
                    <mx:GroupingField name="Territory"/>
                </mx:Grouping>
            </mx:GroupingCollection>
        </mx:dataProvider>

        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumn dataField="Actual"/>
```

```
            <mx:AdvancedDataGridColumn dataField="Estimate"/>
            <mx:AdvancedDataGridColumn dataField="Min Actual"/>
            <mx:AdvancedDataGridColumn dataField="Max Actual"/>
            <mx:AdvancedDataGridColumn dataField="Min Estimate"/>
            <mx:AdvancedDataGridColumn dataField="Max Estimate"/>
        </mx:columns>
    </mx:AdvancedDataGrid>
</mx:Application>
```

Creating a custom summary function

Use the `SummaryRow.summaryObjectFunction` property and the `SummaryField.summaryFunction` property to add custom summary logic to your application. Your custom summary can perform many types of actions, but the result of the summary calculation must be of type Number.

The `SummaryRow.summaryObjectFunction` property defines a callback function that defines an instance of the SummaryObject class that collects the summary data. The AdvancedDataGrid control adds the SummaryObject instance to the data provider to display the summary data in the control. Therefore, you define within the SummaryObject instance the properties to display. In the following example, the callback function adds a property named Territory_Rep. This field appears in the Territory Rep column of the control for the summary row.

The `SummaryField.summaryFunction` property defines a callback function to perform the summary calculation. The callback function must return a Number that contains the summary. The Number is displayed in the summary row in the column that corresponds to the field being summarized.

The following example implements callback functions for the `summaryObjectFunction` and `summaryFunction` properties to calculate a summary of the actual sales revenue for every other row of each territory:

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

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;
            import mx.controls.advancedDataGridClasses.AdvancedDataGridColumn;
            import mx.collections.IViewCursor;
            import mx.collections.SummaryObject;

            include "SimpleFlatData.as"

            // Callback function to create
            // the SummaryObject used to hold the summary data.
            private function summObjFunc():SummaryObject {

                // Define the object containing the summary data.
                var obj:SummaryObject = new SummaryObject();
                // Add a field containing a value for the Territory_Rep column.
                obj.Territory_Rep = "Alternating Reps";

                return obj;
            }

            // Callback function to summarizes
            // every other row of the Actual sales revenue for the territory.
            private function summFunc(cursor:IViewCursor, dataField:String,
                operation:String):Number {

                var oddCount:Number = 0;
                var count:int = 1;
                while (!cursor.afterLast)
```

```
                {
                    if (count % 2 != 0)
                        oddCount += cursor.current["Actual"];

                    cursor.moveNext();
                    count++;
                }

                return oddCount;
            }
        ]]>
    </mx:Script>

    <mx:AdvancedDataGrid id="myADG"
        width="100%" height="100%"
        initialize="gc.refresh();">
        <mx:dataProvider>
            <mx:GroupingCollection id="gc" source="{dpFlat}">
                <mx:Grouping>
                   <mx:GroupingField name="Region"/>
                   <mx:GroupingField name="Territory">
                      <mx:summaries>
                         <mx:SummaryRow summaryObjectFunction="summObjFunc"
                            summaryPlacement="first">
                            <mx:fields>
                           <mx:SummaryField dataField="Actual" summaryFunction="summFunc"/>
                            </mx:fields>
                         </mx:SummaryRow>
                      </mx:summaries>
                   </mx:GroupingField>
                </mx:Grouping>
            </mx:GroupingCollection>
        </mx:dataProvider>

        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumn dataField="Actual"/>
            <mx:AdvancedDataGridColumn dataField="Estimate"/>
        </mx:columns>
    </mx:AdvancedDataGrid>
</mx:Application>
```

Creating column groups

You collect multiple columns under a single column heading by using column groups, as the following example shows:

Region	Territory	Territory Rep	Revenues	
			Actual	Estimate
Southwest	Arizona	Barbara Jennings	38865	40000
Southwest	Arizona	Dana Binn	29885	30000
Southwest	Central California	Joe Smith	29134	30000
Southwest	Nevada	Bethany Pittman	52888	45000
Southwest	Northern California	Lauren Ipsum	38805	40000
Southwest	Northern California	T.R. Smith	55498	40000
Southwest	Southern California	Alice Treu	44985	45000
Southwest	Southern California	Jane Grove	44913	45000

In this example, you use flat data to populate the data grid, and group the Actual and Estimate columns under a single column named Revenues.

To group columns in an AdvancedDataGrid control, you must do the following:

- Use the `AdvancedDataGrid.groupedColumns` property, rather than the `AdvancedDataGrid.columns` property, to specify the columns.
- Use the AdvancedDataGridColumnGroup class to specify the column groups.

The following code implements the AdvancedDataGrid control shown in the previous figure:

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

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            // Import the data used by the AdvancedDataGrid control.
            include "SimpleFlatData.as";
        ]]>
    </mx:Script>

    <mx:AdvancedDataGrid id="myADG"
        dataProvider="{dpFlat}"
        width="100%" height="100%">
        <mx:groupedColumns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumnGroup headerText="Revenues">
                <mx:AdvancedDataGridColumn dataField="Actual"/>
                <mx:AdvancedDataGridColumn dataField="Estimate"/>
            </mx:AdvancedDataGridColumnGroup>
        </mx:groupedColumns>
    </mx:AdvancedDataGrid>
```

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

The `groupedColumns` property contains instances of the AdvancedDataGridColumn class and of the AdvancedDataGridColumn class. Instances of the AdvancedDataGridColumn class appear in the control as stand-alone columns. All the columns specified in an AdvancedDataGridColumnGroup instance appear together as a grouped column.

You can add multiple groups to the control. The following example adds groups named Area and Revenues:

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/ColumnGroupADG2Groups.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            include "SimpleFlatData.as";
        ]]>
    </mx:Script>

    <mx:AdvancedDataGrid id="myADG"
        dataProvider="{dpFlat}"
        width="100%" height="100%">
        <mx:groupedColumns>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumnGroup headerText="Area">
                <mx:AdvancedDataGridColumn dataField="Region"/>
                <mx:AdvancedDataGridColumn dataField="Territory"/>
            </mx:AdvancedDataGridColumnGroup>
            <mx:AdvancedDataGridColumnGroup headerText="Revenues">
                <mx:AdvancedDataGridColumn dataField="Actual"/>
                <mx:AdvancedDataGridColumn dataField="Estimate"/>
            </mx:AdvancedDataGridColumnGroup>
        </mx:groupedColumns>
   </mx:AdvancedDataGrid>
</mx:Application>
```

You can nest groups so that one column group contains multiple column groups, as the following example shows:

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/ColumnGroupADG2NestedGroups.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            include "SimpleFlatData.as";
        ]]>
    </mx:Script>

    <mx:AdvancedDataGrid id="myADG"
        dataProvider="{dpFlat}"
        width="100%" height="100%">
        <mx:groupedColumns>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumnGroup headerText="All Groups">
                <mx:AdvancedDataGridColumnGroup headerText="Area">
                    <mx:AdvancedDataGridColumn dataField="Region"/>
                    <mx:AdvancedDataGridColumn dataField="Territory"/>
```

```
                </mx:AdvancedDataGridColumnGroup>
                <mx:AdvancedDataGridColumnGroup headerText="Revenues">
                    <mx:AdvancedDataGridColumn dataField="Actual"/>
                    <mx:AdvancedDataGridColumn dataField="Estimate"/>
                </mx:AdvancedDataGridColumnGroup>
            </mx:AdvancedDataGridColumnGroup>
        </mx:groupedColumns>
    </mx:AdvancedDataGrid>
</mx:Application>
```

Dragging and dropping columns in the group

By default, you can drag the columns in a group within the group to reposition them. You can also drag the entire group to reposition it in the AdvancedDataGrid control.

To disable dragging of all columns in a group, set the `AdvancedDataGridColumnGroup.childrenDragEnabled` property to `false`. To disable dragging for a single column, set the `AdvancedDataGridColumn.dragEnabled` property to `false`.

Using grouped columns with hierarchical data

You can use column groups with hierarchical data, as well as flat data. The following example modifies the example in the section "Creating a separate column for the navigation tree" on page 293 to group the Actual and Estimates columns under the Revenues group column:

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

  <mx:Script>
    <![CDATA[
          import mx.collections.ArrayCollection;

          include "SimpleHierarchicalData.as";
      ]]>
    </mx:Script>

    <mx:AdvancedDataGrid width="100%" height="100%">
        <mx:dataProvider>
            <mx:HierarchicalData source="{dpHierarchy}"/>
        </mx:dataProvider>
        <mx:groupedColumns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumnGroup headerText="Revenues">
                <mx:AdvancedDataGridColumn dataField="Actual"/>
                <mx:AdvancedDataGridColumn dataField="Estimate"/>
            </mx:AdvancedDataGridColumnGroup>
        </mx:groupedColumns>
    </mx:AdvancedDataGrid>
</mx:Application>
```

Specify a data field for the AdvancedDataGridColumnGroup class

The previous examples for column groups do not specify a data field for the AdvancedDataGridColumnGroup class. However, the AdvancedDataGridColumnGroup class is designed to work with hierarchical data. Therefore, if you specify a data field for the AdvancedDataGridColumnGroup class, it automatically creates a column group for the subfields of that data field.

In the following example, the HierarchicalDataForGroupedColumns.as file defines a hierarchical data set where the Revenues field contains two subfields, Actual and Estimate:

```
[Bindable]
private var dpHierarchy:ArrayCollection = new ArrayCollection([
  {Region:"Southwest", Territory:"Arizona",
      Territory_Rep:"Barbara Jennings",
      Revenues:{Actual:38865, Estimate:40000}},
  {Region:"Southwest", Territory:"Arizona",
      Territory_Rep:"Dana Binn",
      Revenues:{Actual:29885, Estimate:30000}},
  {Region:"Southwest", Territory:"Central California",
      Territory_Rep:"Joe Smith",
      Revenues:{Actual:29134, Estimate:30000}},
  {Region:"Southwest", Territory:"Nevada",
      Territory_Rep:"Bethany Pittman",
      Revenues:{Actual:52888, Estimate:45000}},
  {Region:"Southwest", Territory:"Northern California",
      Territory_Rep:"Lauren Ipsum",
      Revenues:{Actual:38805, Estimate:40000}},
  {Region:"Southwest", Territory:"Northern California",
      Territory_Rep:"T.R. Smith",
      Revenues:{Actual:55498, Estimate:40000}},
  {Region:"Southwest", Territory:"Southern California",
      Territory_Rep:"Alice Treu",
      Revenues:{Actual:44985, Estimate:45000}},
  {Region:"Southwest", Territory:"Southern California",
      Territory_Rep:"Jane Grove",
      Revenues:{Actual:44913, Estimate:45000}}
]);
```

The following example uses this data and specifies the Revenues field as the value of the `AdvancedDataGridColumnGroup.dataField` property, and creates the following output:

Region	Territory	Territory Rep	Revenues	
			Actual	Estimate
Southwest	Arizona	Barbara Jennings	38865	40000
Southwest	Arizona	Dana Binn	29885	30000
Southwest	Central California	Joe Smith	29134	30000
Southwest	Nevada	Bethany Pittman	52888	45000
Southwest	Northern California	Lauren Ipsum	38805	40000
Southwest	Northern California	T.R. Smith	55498	40000
Southwest	Southern California	Alice Treu	44985	45000
Southwest	Southern California	Jane Grove	44913	45000

The following code implements this example:

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/ColumnGroupHierarchData.mxml -->
```

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

    <mx:Script>
        <![CDATA[
            import mx.collections.ArrayCollection;

            include "HierarchicalDataForGroupedColumns.as";
        ]]>
    </mx:Script>

    <mx:AdvancedDataGrid id="myADG"
        width="100%" height="100%"
        defaultLeafIcon="{null}">
        <mx:dataProvider>
            <mx:HierarchicalData source="{dpHierarchy}"/>
        </mx:dataProvider>
        <mx:groupedColumns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumnGroup dataField="Revenues">
                <mx:AdvancedDataGridColumn dataField="Actual"/>
                <mx:AdvancedDataGridColumn dataField="Estimate"/>
            </mx:AdvancedDataGridColumnGroup>
        </mx:groupedColumns>
   </mx:AdvancedDataGrid>
</mx:Application>
```

Using item renderers with the AdvancedDataGrid control

You customize the appearance and behavior of cells in an AdvancedDataGrid control by creating custom item renderers and item editors. Use the same process for assigning item editors and item renderers to columns of the AdvancedDataGrid control as you do with the DataGrid control. For an introduction to item renderers and item editors, see "Using Item Renderers and Item Editors" on page 779 in *Adobe Flex 3 Developer Guide*.

The AdvancedDataGrid control adds capabilities to support item renderers. These capabilities let you perform the following actions:

- Create rows or columns not associated with data in the data provider. For example, you can create summary rows from the data provider.
- Span multiple columns with a renderer.
- Use multiple renderers in the same column. For example, when displaying hierarchical data, you can use a different renderer in the column based on the level of the row in the hierarchy.

Note: *These capabilities support item renderers only; use item editors as you do with the DataGrid control.*

Using an item renderer

To use the item renderer capabilities of the AdvancedDataGrid control, you assign the item renderer to the AdvancedDataGrid control itself, not to a specific column, by using the `AdvancedDataGrid.rendererProviders` property. The following example assigns an item renderer to the Estimate column:

```
<mx:AdvancedDataGrid>
```

```
    <mx:columns>
        <mx:AdvancedDataGridColumn dataField="Region"/>
        <mx:AdvancedDataGridColumn dataField="Territory_Rep" headerText="Territory Rep"/>
        <mx:AdvancedDataGridColumn dataField="Actual"/>
        <mx:AdvancedDataGridColumn dataField="Estimate"/>
    </mx:columns>

    <mx:rendererProviders>
        <mx:AdvancedDataGridRendererProvider
            columnIndex="3"
            columnSpan="1"
            renderer="myComponents.EstimateRenderer"/>
    </mx:rendererProviders>
</mx:AdvancedDataGrid>
```

The `rendererProviders` property contains an Array of AdvancedDataGridRendererProvider instances. Each AdvancedDataGridRendererProvider instance defines the characteristics for a single item renderer.

In the previous example, the AdvancedDataGridRendererProvider instance specifies to use the EstimateRenderer for column 3, where the first column in the control is column 0, and the renderer spans a single column. If you set the `columnSpan` property to 0, the renderer spans all columns in the row.

Rather than using the column index to assign the renderer, you specify an `id` property for the column, and then use the `column` property to associate the renderer with the column, as the following example shows:

```
<mx:AdvancedDataGrid>
    <mx:columns>
        <mx:AdvancedDataGridColumn dataField="Region"/>
        <mx:AdvancedDataGridColumn dataField="Territory_Rep" headerText="Territory Rep"/>
        <mx:AdvancedDataGridColumn dataField="Actual"/>
        <mx:AdvancedDataGridColumn id="estCol" dataField="Estimate"/>
    </mx:columns>

    <mx:rendererProviders>
        <mx:AdvancedDataGridRendererProvider
            column="{estCol}"
            columnSpan="1"
            renderer="myComponents.EstimateRenderer"/>
    </mx:rendererProviders>
</mx:AdvancedDataGrid>
```

The `depth` property lets you associate the renderer with a specific level in the hierarchy of the navigation tree of an AdvancedDataGrid control, where the depth of the top-most node in the navigation tree is 1. The following example associates the renderer with the third level of a navigation tree:

```
    <mx:rendererProviders>
        <mx:AdvancedDataGridRendererProvider
            columnIndex="3"
            depth="3"
            columnSpan="1"
            renderer="myComponents.EstimateRenderer"/>
    </mx:rendererProviders>
```

In the previous example, the AdvancedDataGrid control uses the default renderer for the column until you expand it to the third level of the navigation tree, and then it uses EstimateRenderer component. You use the `depth` property to assign different renderers to the same column, where the `depth` property specifies the renderer to use for each level of the tree.

A renderer can span multiple columns. In the following example, you create a renderer that spans two columns:

```
<mx:AdvancedDataGrid>
    <mx:columns>
        <mx:AdvancedDataGridColumn dataField="Region"/>
```

```
        <mx:AdvancedDataGridColumn dataField="Territory_Rep" headerText="Territory Rep"/>
        <mx:AdvancedDataGridColumn id="actCol" dataField="Actual"/>
        <mx:AdvancedDataGridColumn dataField="Estimate"/>
    </mx:columns>

    <mx:rendererProviders>
        <mx:AdvancedDataGridRendererProvider
            column="{actCol}"
            depth="3"
            columnSpan="2"
            renderer="myComponents.SummaryRenderer"/>
    </mx:rendererProviders>
</mx:AdvancedDataGrid>
```

The previous example uses a single renderer that spans the Actual and Estimate columns to display a combined view of the data for these columns. For an implementation of the SummaryRenderer component, see "Using a renderer to generate column data" on page 315.

The following table describes the properties of the AdvancedDataGridRendererProvider class that you use to configure the renderer:

Property	Description
`column`	The ID of the column for which the renderer is used. If you omit this property, you can use the `columnIndex` property to specify the column.
`columnIndex`	The column index for which the renderer is used, where the first column has an index of 0.
`columnSpan`	The number of columns that the renderer spans. Set this property to 0 to span all columns. The default value is 1.
`dataField`	The data field in the data provider for the renderer. This property is optional.
`depth`	The depth in the tree at which the renderer is used, where the top-most node of the tree has a depth of 1. Use this property when the renderer should be used only when the tree is expanded to a certain depth, not for all nodes in the tree. By default, the control uses the renderer for all levels of the tree.
`renderer`	The renderer.
`rowSpan`	The number of rows that the renderer spans. The default value is 1.

Using a renderer to generate column data

The following example shows the results when an item renderer is used to calculate the value of the Difference column of the AdvancedDataGrid control:

This example defines a column with the `id` of diffCol that is not associated with a data field in the data provider. Instead, you use the `rendererProvider` property to assign an item renderer to the column. The item renderer calculates the difference between the actual and estimate values, and then displays a message based on whether the representative exceeded or missed their estimate.

The following code implements this example:

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

    <mx:Script>
        <![CDATA[
              import mx.collections.ArrayCollection;

              include "SimpleHierarchicalData.as";
        ]]>
    </mx:Script>

    <mx:AdvancedDataGrid width="100%" height="100%">
        <mx:dataProvider>
            <mx:HierarchicalData source="{dpHierarchy}"/>
        </mx:dataProvider>
        <mx:columns>
            <mx:AdvancedDataGridColumn dataField="Region"/>
            <mx:AdvancedDataGridColumn dataField="Territory_Rep"
                headerText="Territory Rep"/>
            <mx:AdvancedDataGridColumn dataField="Actual"/>
            <mx:AdvancedDataGridColumn dataField="Estimate"/>
            <mx:AdvancedDataGridColumn id="diffCol"
                headerText="Difference"/>
        </mx:columns>

        <mx:rendererProviders>
            <mx:AdvancedDataGridRendererProvider column="{diffCol}"
                depth="3" renderer="myComponents.SummaryRenderer"/>
        </mx:rendererProviders>
    </mx:AdvancedDataGrid>
</mx:Application>
```

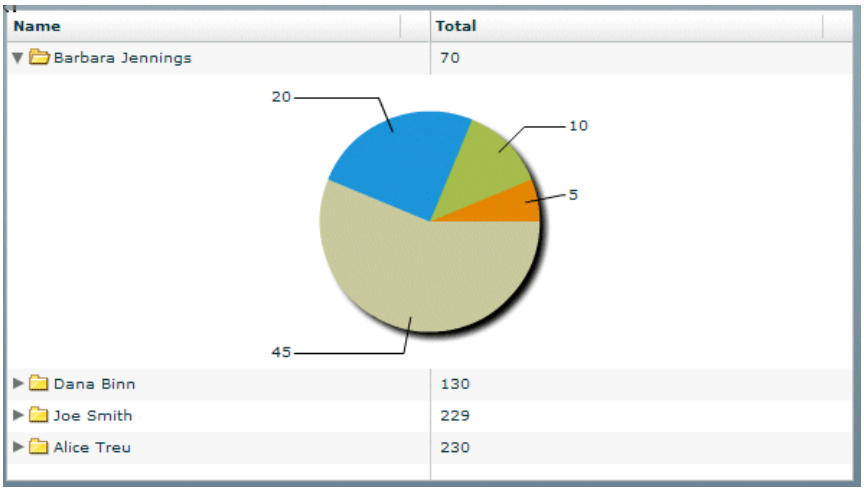
The following code shows the SummaryRenderer component:

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/myComponents/SummaryRenderer.mxml -->
<mx:Label xmlns:mx="http://www.adobe.com/2006/mxml" textAlign="center">
    <mx:Script>
        <![CDATA[

        override public function set data(value:Object):void
        {
            // Calculate the difference.
            var diff:Number =
                Number(value["Actual"]) - Number(value["Estimate"]);
            if (diff < 0)
            {
                // If Estimate was greater than Actual,
                // display results in red.
                setStyle("color", "red");
                text = "Undersold by " + usdFormatter.format(diff);
            }
            else
            {
```

```
                // If Estimate was less than Actual,
                // display results in green.
                setStyle("color", "green");
                text = "Exceeded estimate by " + usdFormatter.format(diff);
            }
        }
        ]]>
    </mx:Script>

    <mx:CurrencyFormatter id="usdFormatter" precision="2"
        currencySymbol="$" decimalSeparatorFrom="."
        decimalSeparatorTo="." useNegativeSign="true"
        useThousandsSeparator="true" alignSymbol="left"/>
</mx:Label>
```

Using an item renderer that spans an entire row

You can use an item renderer with a hierarchical data set to display an entire row of data. The following example shows a PieChart control that displays the data from the detail field of the hierarchical data set. Each row contains detailed information about sales revenues for the representative, which is rendered as a segment of the pie chart:



The following code implements this example:

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

  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;

      [Bindable]
      private var dpHierarchy:ArrayCollection= new ArrayCollection([
        {name:"Barbara Jennings", region: "Arizona", total:70, children:[
          {detail:[{amount:5},{amount:10},{amount:20},{amount:45}]}]},
        {name:"Dana Binn", region: "Arizona", total:130,  children:[
          {detail:[{amount:15},{amount:25},{amount:35},{amount:55}]}]},
        {name:"Joe Smith", region: "California", total:229,  children:[
          {detail:[{amount:26},{amount:32},{amount:73},{amount:123}]}]},
        {name:"Alice Treu", region: "California", total:230, children:[
          {detail:[{amount:159},{amount:235},{amount:135},{amount:155}]}
```

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

  <mx:AdvancedDataGrid id="myADG"
    width="100%" height="100%"
    variableRowHeight="true">
    <mx:dataProvider>
       <mx:HierarchicalData source="{dpHierarchy}"/>
    </mx:dataProvider>
    <mx:columns>
        <mx:AdvancedDataGridColumn dataField="name" headerText="Name"/>
        <mx:AdvancedDataGridColumn dataField="total" headerText="Total"/>
    </mx:columns>

    <mx:rendererProviders>
        <mx:AdvancedDataGridRendererProvider
            dataField="detail"
            renderer="myComponents.ChartRenderer"
            columnIndex="0"
            columnSpan="0"/>
    </mx:rendererProviders>
  </mx:AdvancedDataGrid>
</mx:Application>
```

The following code renders the ChartRenderer.mxml component:

```
<?xml version="1.0"?>
<!-- dpcontrols/adg/myComponents/ChartRenderer.mxml -->
<mx:VBox height="200" width="100%" xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:PieChart dataProvider="{data.detail}"
        width="100%"
        height="100%"
        showDataTips="true">
        <mx:series>
            <mx:PieSeries labelPosition="callout" field="amount"  />
        </mx:series>
    </mx:PieChart>
</mx:VBox>
```

You can modify this example to display the PieChart control in a column. In the following example, you add a Detail column to the control, and then specify that the item renderer is for column 2 of the control:

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

  <mx:Script>
    <![CDATA[
      import mx.collections.ArrayCollection;

      [Bindable]
      private var dpHierarchy:ArrayCollection= new ArrayCollection([
        {name:"Barbara Jennings", region: "Arizona", total:70, children:[
          {detail:[{amount:5},{amount:10},{amount:20},{amount:45}]}]},
        {name:"Dana Binn", region: "Arizona", total:130,  children:[
          {detail:[{amount:15},{amount:25},{amount:35},{amount:55}]}]},
        {name:"Joe Smith", region: "California", total:229,  children:[
          {detail:[{amount:26},{amount:32},{amount:73},{amount:123}]}]},
        {name:"Alice Treu", region: "California", total:230, children:[
          {detail:[{amount:159},{amount:235},{amount:135},{amount:155}]}
        ]}
```

```
      ]);
    ]]>
  </mx:Script>

  <mx:AdvancedDataGrid id="myADG"
    width="100%" height="100%"
    variableRowHeight="true">
    <mx:dataProvider>
       <mx:HierarchicalData source="{dpHierarchy}"/>
    </mx:dataProvider>
    <mx:columns>
        <mx:AdvancedDataGridColumn dataField="name" headerText="Name"/>
        <mx:AdvancedDataGridColumn dataField="total" headerText="Total"/>
        <mx:AdvancedDataGridColumn dataField="detail" headerText="Detail"/>
    </mx:columns>

    <mx:rendererProviders>
        <mx:AdvancedDataGridRendererProvider
            dataField="detail"
            renderer="myComponents.ChartRenderer"
            columnIndex="2"/>
    </mx:rendererProviders>
  </mx:AdvancedDataGrid>
</mx:Application>
```

Keyboard navigation

The following keystrokes are built into the AdvancedDataGrid control to let users navigate the control:

Keyboard action	Key assignments
Standard	• Use the Up and Down Arrow keys to scroll vertically by one row. • Type characters to do incremental type-ahead look-ups in the first column. • Use the Home and End keys to move to the first and last rows. • Use the Page Up and Page Down keys to move to the last visible row and then scroll vertically by the number of rows specified by the `rowcount` property.
In editable mode	• Use the Tab and Shift+Tab keys to move to the next and previous cells, and start editing. • Use the Escape or Control+Period keys to cancel editing. • Use the Enter key based on the value of the `AdvancedDataGridColumn.editorUsesEnterKey` property: • If `editorUsesEnterKey=true`, the Enter key is considered to be part of the input text. • If `editorUsesEnterKey=false`, if single-line editor, the Enter key saves the item and moves down a row. • If you use the Tab key to move focus to the AdvancedDataGrid control, focus is placed on the position of the last edited item. • Use the Control click keys to select multiple discontiguous items.
For the expandable tree	• Use the Multiply key on the number pad (asterisk) to open or close all nodes under the current node, with no animation. • Use the Add key on the number pad or the Space key to open nodes with animation. • Use the Subtract key on the number pad to close nodes with animation. • Use the Shift+Right Arrow key to open a closed node, or select the parent node. • Use the Shift+Left Arrow to close open nodes. • Use the Up and Down Arrow keys to move up and down one row.

Keyboard action	Key assignments
For column headers	• Use the arrow keys to move focus to a column header. • The first Space key pressed when a column header has focus sorts the column in descending order. Each subsequent press of the Space key toggles the sort order between ascending and descending order. • Use the Control+Space keys when a header has focus to sort multiple columns. • Use the Left Arrow and Right Arrow keys to move to the previous or next header, without column wrapping.
To scroll vertically and horizontally in pages	• The first Page Up or Page Down key pressed move to the first or last visible row. The next Page Up or Page Down key pressed scrolls vertically up or down by a page, which corresponds to the number of visible rows. • Use the Shift+Page Up and Shift+Page Down keys to move to first or last visible column.
To move focus out of AdvancedDataGrid while editing is in progress	When you set `AdvancedDataGrid.editable=true` or `AdvancedDataGridColumn=true`: • Use the Tab key to move focus to the AdvancedDataGrid control. • Use the Tab key again to move focus to the next component. When focus is on the AdvancedDataGrid control: • Use the Left, Right, Up, and Down Arrow keys to move between cells. • Use the Shift+Home and Shift+End keys to move to the first and last column in current row. • Cells are only selected by default, they are not editable. • Press the F2 key to make a cell editable. When editing the cell: • Use the arrow keys to move the cursor in the editing area. • Use the Escape key to cancel editing. • Use the Enter key to commit editing changes. When you set `AdvancedDataGrid.editable=false` or `AdvancedDataGridColumn=false`: • Same behavior as when `editable=true`, but the F2 key does not make the cell editable. • Use the Home and End keys to move to first and last rows. • Use the Page Up and Page Down keys to move to the previous and next pages.

Chapter 7: Creating OLAP Data Grids

The OLAPDataGrid control lets you aggregate large amounts of data for easy display and interpretation. The control supports a two-dimensional grid layout, where the columns and rows of the control can display a single level of information, or you can create hierarchical rows and columns to display more complex data.

The OLAPDataGrid controls is a subclass of the AdvancedDataGrid control, and therefore supports many of the features of the AdvancedDataGrid control. For more information on the AdvancedDataGrid control, see "Using the AdvancedDataGrid Control" on page 275.

Topics

About OLAP data grids 322
Creating an OLAP schema 330
Creating OLAP queries 333
Writing a query for a simple OLAP cube 338
Writing a query for a complex OLAP cube 344
Creating a multidimensional axis in an OLAPDataGrid control 348
Configuring the display of an OLAPDataGrid 352

About OLAP data grids

When working with large amounts of data, you can quickly get overwhelmed with the scope and size of the data. For example, you collect sales information for different products, in different regions, and for different customers in a typical two-dimensional spreadsheet. That spreadsheet could easily contain hundreds of rows and tens or even hundreds of columns. Extracting useful information for such a large data collection can be difficult, and trying to identify trends or other patterns in the data can be even harder.

Data visualization is a technique for examining large amounts of data in a compact format. One type of data visualization technique is to use a chart, such as a bar, column, or pie chart. Adobe® Flex™ supports many types of charts. For more information, see "Introduction to Charts" on page 2.

Another data visualization technique is to aggregate the data in a compact format, such as in an OLAP (online analytical processing) data grid. An OLAP data grid is similar to a pivot table in Microsoft Excel. An OLAP data grid displays data aggregations in a two-dimensional grid of rows and columns, like a spreadsheet, but the data is condensed based on your aggregation settings.

Note: *While the Flex OLAP data grid is similar to a pivot table, it provides a much greater set of features for aggregating data.*

For example, you collect sales information on a server in a flat data structure of records, where each record contains information for a single customer transaction, for a single product, in a single quarter. The following code shows the format of this flat data:

```
data:Object = {
    customer:"AAA",
    product:"ColdFusion",
    year:"2007",
    quarter:"Q1"
    revenue: "100.00"
}
```

For a large company with hundreds of customers and tens or hundreds of products, this table could easily contain several thousand rows. Rather than display this information in a standard spreadsheet, you download your data to a Flex application to aggregate sales data by product and quarter, and then display the aggregated data in an OLAP data grid. From this data aggregation, you can determine trends in sales of each product over time.

About the OLAPDataGrid control

You use the Flex OLAPDataGrid control to display an OLAP data grid. The following image shows the OLAPDataGrid control displaying the aggregated sales information for product and quarter:

Product	Quarter			
	Q1	Q2	Q3	Q4
ColdFusion	1485	300	785	430
Flex	420	1430	310	730
DreamWeaver	980	350	1990	590
Flash	650	1190	1090	2250

Like all Flex data grid controls, the OLAPDataGrid control is designed to display data in a two-dimensional representation of rows and columns.

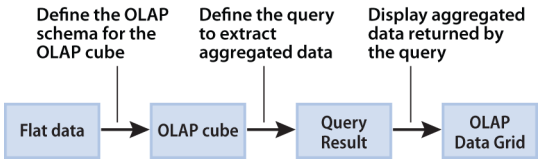
You can modify this example to compare quarterly sales from two different years, as the following example shows:

Product	Year	Quarter						
	2006				2007			
	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4
ColdFusion	770	125	375	215	715	175	410	215
Flex	210	760	210	430	210	670	100	300
DreamWeaver	470	175	840	320	510	175	1150	270
Flash	280	570	670	1010	370	620	420	1240

In the previous figure, the columns of the OLAPDataGrid control show a hierarchy of information for year and quarter. You can add multiple-level hierarchies for both the columns and the rows of the control.

About creating an OLAP data grid

The following figure shows the data flow that you use to aggregate your data for display in the OLAPDataGrid control:



The following steps describe this process in more detail:

1 Start with flat data, which is typically data arranged as a set of records where each record contains the same data fields. For example, you might start with flat data from a spreadsheet, or with data from a table of a relational database.

The following code shows an example format of flat data:

```
data:Object = {
    customer:"AAA",
    product:"ColdFusion",
    quarter:"Q1"
    revenue: "100.00"
}
```

2 Define an OLAP schema that describes how your data gets transformed from a flat representation into an OLAP cube. An OLAP schema defines the representation of your flat data in an OLAP cube, and defines how to aggregate your data for an OLAP query.

An OLAP cube is analogous to a table in a relational database. But whereas a table typically has two dimensions (row and column), an OLAP cube can have any number of dimensions. In this example, the cube has three dimensions: customer, product, and quarter. Every possible set of values for customer, product, and quarter defines a unique point in the cube. The value at each point in the cube is the sales revenue for that set of values of customer, product, and quarter.

3 Create queries to extract aggregated data from the OLAP cube for display in the OLAPDataGrid control.

After your data is in an OLAP cube, you write queries to extract aggregated data for display in the OLAPDataGrid control. You can write multiple queries to create different types of data aggregations.

4 Use the query results as input to the OLAPDataGrid control to display the results.

About OLAP cubes

An OLAP cube can have any number of dimensions. In its simplest form, the *dimensions* of an OLAP cube correspond to a field of the flat data set. For example, you have flat data that contains three fields:

```
data:Object = {
    product:"ColdFusion"
    quarter :"Q1"
    revenue: "100.00",
}
```

The data fields of each record can contain the following values:

- The product field can have the values: ColdFusion, Flex, Dreamweaver, and Illustrator.
- The quarter field can have the values: Q1, Q2, Q3, and Q4.
- The revenue field contains the sales, in dollars, of the product for the quarter.

To aggregate your data, you create an OLAP cube with two dimensions: quarter and product. The value along each dimension of the cube is called a *member.* For example, the product dimension of the cube has the following members: ColdFusion, Flex, Dreamweaver, and Illustrator. For the quarter dimension, the members are Q1, Q2, Q3, and Q4.

The value at any point in the cube defined by the two dimensions is called a *measure* of the cube. For example, the measure at the point in the cube defined by (Q1, ColdFusion) is 100.00. A schema can define one or more measures for a single point in the OLAP cube.

Flex supports only numeric values for the measure of a cube. The advantage of numeric values is that they can be easily aggregated for display in the OLAPDataGrid control. Some typical aggregation types include sum, average, minimum, and maximum. For example, you specify the aggregation method of the revenue measure as `SUM`. You then extract sales information from the cube for ColdFusion. The aggregated sales data contains the sum of all ColdFusion sales for each quarter.

About OLAP schemas

To convert flat data into an OLAP cube, you create an OLAP *schema* that defines the dimensions of the cube, the fields of the flat data that supply the members along each dimension, and the fields of the flat data that supply the measure for any point in the cube.

For example, you have the following flat data that contains sales records:

```
data:Object = {
    customer:"AAA",
    product:"ColdFusion",
    quarter:"Q1"
    revenue: "100.00"
}
```

The following example shows the definition for an OLAPCube that includes the definition of the OLAP schema used to represent this data in the cube. This schema defines a three-dimensional OLAP cube based on the customer, product, and quarter fields of the data.

```
<mx:OLAPCube name="FlatSchemaCube"
    dataProvider="{flatData}"
    id="myMXMLCube"
    complete="runQuery(event);">

    <mx:OLAPDimension name="CustomerDim">
        <mx:OLAPAttribute name="Customer" dataField="customer"/>
        <mx:OLAPHierarchy name="CustomerHier" hasAll="true">
            <mx:OLAPLevel attributeName="Customer"/>
        </mx:OLAPHierarchy>
    </mx:OLAPDimension>

    <mx:OLAPDimension name="ProductDim">
        <mx:OLAPAttribute name="Product" dataField="product"/>
        <mx:OLAPHierarchy name="ProductHier" hasAll="true">
            <mx:OLAPLevel attributeName="Product"/>
        </mx:OLAPHierarchy>
    </mx:OLAPDimension>

    <mx:OLAPDimension name="QuarterDim">
        <mx:OLAPAttribute name="Quarter" dataField="quarter"/>
        <mx:OLAPHierarchy name="QuarterHier" hasAll="true">
            <mx:OLAPLevel attributeName="Quarter"/>
        </mx:OLAPHierarchy>
    </mx:OLAPDimension>

    <mx:OLAPMeasure name="Revenue"
        dataField="revenue"
        aggregator="SUM"/>
</mx:OLAPCube>
```

Notice that in this schema:

- All dimensions are defined first, and then all measures.

- The first line of each dimension associates a data field of the flat data with an OLAPAttribute instance. You then use the `OLAPLevel.attributeName` property to associate the attribute with a level of the dimension to populate the members of the dimension. For example, in this schema you populate the Customer level of the CustomerDim dimension with the data from the customer field of the data.
- A dimension of an OLAP schema always contains a hierarchy of one or more levels. In this schema, the hierarchy of each dimension contains only a single level corresponding to a field of the flat data. This is the simplest form of a dimension. Other schemas could define multiple levels in a hierarchy to create a complex dimension. For more information, see "Creating an OLAP schema" on page 330.
- A measure definition specifies the data field of the flat data that contains the value for each point in the OLAP cube, and how the measure is aggregated by an OLAP query. In this example, you aggregate revenue by summing it. That means all queries of this OLAP cube will return revenue summations. Other types of aggregation methods include maximum, minimum, and average.
- This definition of the OLAPCube specifies an event handler for the `complete` event. You cannot invoke a query on the cube until it completes initialization, which is signalled by the cube when it dispatches the `complete` event.

Based on the requirements of your application, you might create multiple OLAP cubes from the same flat data set, where each cube uses a different schema to create its own arrangement of dimensions and measures. For more information and examples of OLAP schemas, see "Creating an OLAP schema" on page 330.

About OLAP queries

OLAP queries extract aggregated data from an OLAP cube for display in an OLAPDataGrid control. The query specifies the dimensions that define the characteristics of the query, and the measure or measures aggregated to create the query results.

In the previous section, you defined an OLAP schema for sales information where the schema defines CustomerDim, ProductDim, and QuarterDim dimensions, and a single measure for revenue aggregated by the `SUM` aggregation method. Therefore, you can create a query to sum revenue by the following criteria:

- Product for each quarter
- Customer for each product
- Customer for each quarter
- Any other combination of members from each dimension

You construct a query in ActionScript as an instance of the OLAPQuery class, and then execute the query by calling the `OLAPCube.execute()` method, which returns an instance of the AsyncToken class.

A query is required to have two axes, a row axis and a column axis, of type IOLAPQueryAxis. The row axis defines the data aggregation information for each row of the OLAPDataGrid control, and the column axis defines the data aggregation information for each column of the control.

You use the OLAPSet class to specify the data aggregation information for each axis, as the following example shows:

```
// Create an instance of OLAPQuery to represent the query.
var query:OLAPQuery = new OLAPQuery;

// Get the row axis from the query instance.
var rowQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.ROW_AXIS);
// Create an OLAPSet instance to configure the axis.
var productSet:OLAPSet = new OLAPSet;
// Add the Product to the row to aggregate data
// by the Product dimension.
productSet.addElements(
    cube.findDimension("ProductDim").findAttribute("Product").children);
// Add the OLAPSet instance to the axis.
rowQueryAxis.addSet(productSet);
```

```
// Get the column axis from the query instance, and configure it
// to aggregate the columns by the Quarter dimension.
var colQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.COLUMN_AXIS);
var quarterSet:OLAPSet= new OLAPSet;
quarterSet.addElements(
    cube.findDimension("QuarterDim").findAttribute("Quarter").children);
colQueryAxis.addSet(quarterSet);

// Execute the query.
var token:AsyncToken = cube.execute(query);
// Set up handlers for the query results.
token.addResponder(new AsyncResponder(showResult, showFault));
```

Notice that in this query:

- You initialize each axis by calling the `OLAPQuery.getAxis()` method.
- You configure each OLAPSet instance by calling the `OLAPSet.addElements()` method to specify the information used to populate the axis.
- You set up two functions to handle the query result defined by the AsyncToken class. In this example, the function `showResult()` handles the query results when the query succeeds, and the function `showFault()` handles any errors detected during query execution. For more information on using the AsyncToken class, see "Executing a query and returning the results to an OLAPDataGrid control" on page 336.

For more information and examples of OLAP queries, see "Creating OLAP queries" on page 333.

The differences between the OLAPDataGrid and the AdvancedDataGrid control

You use the OLAPDataGrid control to display the results of an OLAP query. The OLAPDataGrid control is a subclass of the AdvancedDataGrid control and inherits much of its functionality. However, because of the way you pass data to the OLAPDataGrid control, it has several differences from the AdvancedDataGrid control:

- Column dragging is not allowed in the OLAPDataGrid control.
- You cannot edit cells in the OLAPDataGrid control because cell data is a result of a query and does not correspond to a single data value in the OLAP cube.
- You cannot sort columns by clicking on headers in the OLAPDataGrid control. Sorting is supported at the dimension level so that you can change the order of members of that dimension.

You populate an OLAPDataGrid control with data by setting its data provider to an instance of the OLAPResult class, which contains the results of an OLAP query. For a complete example that uses this control, see "Example using the OLAPDataGrid control" on page 327.

Example using the OLAPDataGrid control

The following example shows the complete code for creating the OLAPDataGrid control that appears in the section "About OLAP data grids" on page 322. This example aggregates product sales by quarter. The example uses flat data from the dataIntro.as file, which contains sales records in the following format:

```
data:Object = {
    customer:"AAA",
    product:"ColdFusion",
    quarter:"Q1"
    revenue: "100.00"
}
```

The following code implements this example:

```
<?xml version="1.0"?>
```

```
<!-- olapdatagrid/OLAPDG_Intro.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
        creationComplete="creationCompleteHandler();">

    <mx:Script>
      <![CDATA[
        import mx.rpc.AsyncResponder;
        import mx.rpc.AsyncToken;
        import  mx.messaging.messages.ErrorMessage;
        import mx.olap.OLAPQuery;
        import mx.olap.OLAPSet;
        import mx.olap.IOLAPQuery;
        import mx.olap.IOLAPQueryAxis;
        import mx.olap.IOLAPCube;
        import mx.olap.OLAPResult;
        import mx.events.CubeEvent;
        import mx.controls.Alert;
        import mx.collections.ArrayCollection;

        include "dataIntro.as"

        private function creationCompleteHandler():void {
            // You must initialize the cube before you
            // can execute a query on it.
            myMXMLCube.refresh();
        }

        // Create the OLAP query.
        private function getQuery(cube:IOLAPCube):IOLAPQuery {
            // Create an instance of OLAPQuery to represent the query.
            var query:OLAPQuery = new OLAPQuery;

            // Get the row axis from the query instance.
            var rowQueryAxis:IOLAPQueryAxis =
                query.getAxis(OLAPQuery.ROW_AXIS);
            // Create an OLAPSet instance to configure the axis.
            var productSet:OLAPSet = new OLAPSet;
            // Add the Product to the row to aggregate data
            // by the Product dimension.
            productSet.addElements(
                cube.findDimension("ProductDim").findAttribute("Product").children);
            // Add the OLAPSet instance to the axis.
            rowQueryAxis.addSet(productSet);

            // Get the column axis from the query instance, and configure it
            // to aggregate the columns by the Quarter dimension.
            var colQueryAxis:IOLAPQueryAxis =
                query.getAxis(OLAPQuery.COLUMN_AXIS);
            var quarterSet:OLAPSet= new OLAPSet;
            quarterSet.addElements(
                cube.findDimension("QuarterDim").findAttribute("Quarter").children);
            colQueryAxis.addSet(quarterSet);

            return query;
        }

        // Event handler to execute the OLAP query
        // after the cube completes initialization.
        private function runQuery(event:CubeEvent):void {
            // Get cube.
            var cube:IOLAPCube = IOLAPCube(event.currentTarget);
```

```
            // Create a query instance.
            var query:IOLAPQuery = getQuery(cube);
            // Execute the query.
            var token:AsyncToken = cube.execute(query);
            // Set up handlers for the query results.
            token.addResponder(new AsyncResponder(showResult, showFault));
        }

        // Handle a query fault.
        private function showFault(error:ErrorMessage, token:Object):void {
            Alert.show(error.faultString);
        }

        // Handle a successful query by passing the query results to
        // the OLAPDataGrid control..
        private function showResult(result:Object, token:Object):void {
            if (!result) {
                Alert.show("No results from query.");
                return;
            }
            myOLAPDG.dataProvider= result as OLAPResult;
        }
      ]]>
    </mx:Script>

    <mx:OLAPCube name="FlatSchemaCube"
        dataProvider="{flatData}"
        id="myMXMLCube"
        complete="runQuery(event);">

        <mx:OLAPDimension name="CustomerDim">
            <mx:OLAPAttribute name="Customer" dataField="customer"/>
            <mx:OLAPHierarchy name="CustomerHier" hasAll="true">
                <mx:OLAPLevel attributeName="Customer"/>
            </mx:OLAPHierarchy>
        </mx:OLAPDimension>

        <mx:OLAPDimension name="ProductDim">
            <mx:OLAPAttribute name="Product" dataField="product"/>
            <mx:OLAPHierarchy name="ProductHier" hasAll="true">
                <mx:OLAPLevel attributeName="Product"/>
            </mx:OLAPHierarchy>
        </mx:OLAPDimension>

        <mx:OLAPDimension name="QuarterDim">
            <mx:OLAPAttribute name="Quarter" dataField="quarter"/>
            <mx:OLAPHierarchy name="QuarterHier" hasAll="true">
                <mx:OLAPLevel attributeName="Quarter"/>
            </mx:OLAPHierarchy>
        </mx:OLAPDimension>

        <mx:OLAPMeasure name="Revenue"
            dataField="revenue"
            aggregator="SUM"/>
    </mx:OLAPCube>

    <mx:OLAPDataGrid id="myOLAPDG" width="100%" height="100%"/>
</mx:Application>
```

Creating an OLAP schema

An OLAP schema defines the representation of your flat data in an OLAP cube, and defines how to aggregate your data for an OLAP query. You typically define your schema in MXML. While you can construct a schema programmatically in ActionScript, that method requires much more coding than MXML.

The following table describes the classes and interfaces that you use to define an OLAP schema:

Class	Interface	Description
OLAPSchema	IOLAPSchema	The schema instance.
OLAPCube	IOLAPCube	The OLAP cube created by the schema.
OLAPDimension	IOLAPDimension	A dimension of the schema.
OLAPAttribute	IOLAPAttribute	An attribute of a dimension
OLAPHierarchy	IOLAPHierarchy	A hierarchy of a dimension.
OLAPLevel	IOLAPLevel	A level of a hierarchy.
OLAPMeasure	IOLAPMeasure	A measure of a dimension.

General form of a schema definition

In MXML, you can define an OLAP schema as part of the definition of an OLAPCube instance. Typically, you set any properties of the OLAPCube instance as tag attributes, and set the `OLAPCube.dimensions` and `OLAPCube.measures` properties as child tags, as the following example shows:

```
<mx:OLAPCube name="FlatSchemaCube"
    dataProvider="{flatData}"
    id="myMXMLCube"
    complete="runQuery(event);">

    <!-- Define dimensions. -->
    <mx:OLAPDimension ... />

    <mx:OLAPDimension ... />
    ...

    <!-- Define measures. -->
    <mx:OLAPMeasure ... />
    ...
</mx:OLAPCube>
```

The order of the OLAPDimension and OLAPMeasure definitions is not important, but you should not place OLAPMeasure definitions between OLAPDimension definitions.

Specifying the aggregation method for a measure

As part of creating an OLAP schema, you specify the data field that provides the value of the measure for each point in the cube. The measure corresponds to the data value at that point in the OLAP cube. For example, if you define a schema for sales information, you might specify as a measure of the cube the revenue for a product, and the aggregation type as `SUM`.

A schema can define one or more measures for a single point in the OLAP cube. The first measure in the schema is called the *default measure*, and is the measure returned by an OLAP query when you do not explicitly specify the measure to return. For more information on selecting a specific measure, see "Creating a query using a nondefault measure" on page 349.

The following example shows a section of an MXML schema definition that specifies two measures for the schema:

```
<mx:OLAPMeasure name="Revenue" dataField="revenue" aggregator="SUM"/>
<mx:OLAPMeasure name="Cost" dataField="cost" aggregator="SUM"/>
```

When creating a schema, you specify the name of the measure, the data field in the input flat data that contains the data for the measure, and an aggregation method of the measure. In this example, the aggregation method is `SUM`. That means when you create an OLAP query for a measure, the query sums all revenue fields to generate the values displayed by the OLAPDataGrid control. You could use this schema to define a cube so that you can total sales and cost information for products, regions, and other characteristics of your data.

To aggregate the revenue data using a different aggregation method, such as average or maximum, create another schema to define a second OLAP cube. The following example shows a section of another MXML schema definition that specifies the aggregation method as `MAX` for the sales data and `MIN` for the cost data:

```
<mx:OLAPMeasure name="Revenue" dataField="revenue" aggregator="MAX"/>
<mx:OLAPMeasure name="Cost" dataField="cost" aggregator="MIN"/>
```

You should take care in how you define your schema because it can limit the types of queries that you can run on it, or the level of detail at which you can create aggregations.

Creating a schema dimension

An OLAPSchema instance can define any number of dimensions, limited only by your input data. A dimension can be a simple dimension with a single level, or it can be a complex dimension with multiple levels. The dimension of a schema always has the same basic form:

```
<mx:OLAPDimension name="QuarterDim">
    <mx:OLAPAttribute name="Quarter" dataField="quarter"/>
    ...
    <mx:OLAPHierarchy name="QuarterHier" hasAll="true">
        <mx:OLAPLevel attributeName="Quarter"/>
        ...
    </mx:OLAPHierarchy>
</mx:OLAPDimension>
```

Notice that in this schema:

- The dimension first uses OLAPAttribute class to specify the data fields of the input data set used to populate the members of the dimension.

The dimension defines an instance of the OLAPHierarchy class. Therefore, a dimension is always assumed to contain a hierarchy of levels, even if that hierarchy contains only a single level.

- The dimension specifies one or more instances of the OLAPLevel class to associate a field of the input with the dimension.

Defining a dimension that contains a single level

To create a simple dimension (a dimension that contains a single level), you define a dimension hierarchy, as the following example shows:

```
<mx:OLAPDimension name="CustomerDim">
    <mx:OLAPAttribute name="Customer" dataField="customer"/>
    <mx:OLAPHierarchy name="CustomerHier" hasAll="true">
        <mx:OLAPLevel attributeName="Customer"/>
    </mx:OLAPHierarchy>
```

```
</mx:OLAPDimension>
```

In this example, the customer field of the input data defines the entire measure of the dimension.

Defining a dimension that contains multiple levels

Instead of creating a separate dimension for each field of your data, you can choose to group related data fields along a single dimension. For example, your data might contain several fields related to the time of a transaction, such as month, quarter, and year. Or your data might contain multiple fields associated with the geographical area of a transaction, such as region, state, province, or country.

The TimeDim dimension in the following example defines hierarchical dimensions that contain two levels, one for year and one for quarter:

```
<mx:OLAPCube name="FlatSchemaCube"
    dataProvider="{flatData}"
    id="myMXMLCube">

    <mx:OLAPDimension name="TimeDim">
        <mx:OLAPAttribute name="Year" dataField="year"/>
        <mx:OLAPAttribute name="Quarter" dataField="quarter"/>
        <mx:OLAPHierarchy name="Time-PeriodHier" hasAll="true">
            <mx:OLAPLevel attributeName="Year"/>
            <mx:OLAPLevel attributeName="Quarter"/>
        </mx:OLAPHierarchy>
    </mx:OLAPDimension>

    <mx:OLAPMeasure name="Revenue"dataField="revenue" aggregator="SUM"/>
</mx:OLAPCube>
```

With this schema definition, you can aggregate data for all time, for individual years, for individual quarters, or for individual quarters of a specific year.

Notice that the TimeDim dimension contains a Year level, the most general level, and a Quarter level, the more detailed level. The first level in the hierarchy typically defines the most general level, and each subsequent level provides a greater level of detail.

If your data contained a month field with the month of the year for a transaction, you can add the Month level to the TimeDim dimension. Since month is a more detailed measure of time than quarter, add the Month level after the Quarter level, as the following example shows:

```
<mx:OLAPDimension name="TimeDim">
    <mx:OLAPAttribute name="Year" dataField="year"/>
    <mx:OLAPAttribute name="Quarter" dataField="quarter"/>
    <mx:OLAPAttribute name="Month" dataField="month"/>
    <mx:OLAPHierarchy name="Time-PeriodHier" hasAll="true">
        <mx:OLAPLevel attributeName="Year"/>
        <mx:OLAPLevel attributeName="Quarter"/>
        <mx:OLAPLevel attributeName="Month"/>
    </mx:OLAPHierarchy>
</mx:OLAPDimension>
```

The advantage of creating a hierarchical schema is that you can write queries to extract top-level data aggregations or to drill down into the data to extract more granular data. For the previous schema definition, you can write an OLAP query to aggregate data by the most general field, such as year, or drill down to obtain more granular results by aggregating your data by quarter in each year, by month in each quarter, or by month in each year.

Creating a default member in a schema

Most of the OLAP schemas shown so far have been defined in the following form:

```
<mx:OLAPDimension name="TimeDim">
    <mx:OLAPAttribute name="Year" dataField="year"/>
    <mx:OLAPAttribute name="Quarter" dataField="quarter"/>
    <mx:OLAPHierarchy name="Time-PeriodHier" hasAll="true">
        <mx:OLAPLevel attributeName="Year"/>
        <mx:OLAPLevel attributeName="Quarter"/>
    </mx:OLAPHierarchy>
</mx:OLAPDimension>
```

In this schema, the hierarchy sets the `OLAPHierarchy.hasAll` property to `true` to create a default member for the hierarchy. The default member is created automatically for the hierarchy and contains an aggregation of all levels in the hierarchy. In the previous schema, the default member contains an aggregation of the measure of the schema for all years and all quarters.

The default member is used by OLAP queries when you do not specify any criteria to aggregate the dimension. For example, your OLAP schema contains a ProductDim, TimeDim, and CustomerDim dimension. You then write a query to extract data by product and customer, but omit any specification in the query for time. The OLAP query automatically uses the default member of the dimension to aggregate the information for the TimeDim dimension.

The default value of the `OLAPHierarchy.hasAll` property is `true`. If you set it to `false`, the OLAP query uses the first level in the hierarchy to aggregate the dimension. The following schema sets the `hasAll` property to `false`:

```
<mx:OLAPDimension name="TimeDim">
    <mx:OLAPAttribute name="Year" dataField="year"/>
    <mx:OLAPAttribute name="Quarter" dataField="quarter"/>
    <mx:OLAPHierarchy name="Time-PeriodHier" hasAll="false">
        <mx:OLAPLevel attributeName="Year"/>
        <mx:OLAPLevel attributeName="Quarter"/>
    </mx:OLAPHierarchy>
</mx:OLAPDimension>
```

Because the `hasAll` property is `false`, an OLAP query automatically aggregates the data in the dimension by the Year level when you omit the TimeDim dimension from the query.

By default, the OLAP cube adds a member to the dimension named `(All)` to represent the default member. You can use the `OLAPHierarchy.allLevelName` property to specify a different name, as the following example shows:

```
<mx:OLAPDimension name="TimeDim">
    <mx:OLAPAttribute name="Year" dataField="year" allLevelname="AllTime"/>
    <mx:OLAPAttribute name="Quarter" dataField="quarter"/>
    <mx:OLAPHierarchy name="Time-PeriodHier" hasAll="true">
        <mx:OLAPLevel attributeName="Year"/>
        <mx:OLAPLevel attributeName="Quarter"/>
    </mx:OLAPHierarchy>
</mx:OLAPDimension>
```

When you execute an OLAP query, you choose whether or not to include the default member in the query results. For more information, see "Using the default member in a query" on page 337.

Creating OLAP queries

You use an OLAP query to extract data from an OLAP cube for display by the OLAPDataGrid control. An OLAP query defines a result set in a two-dimensional table of rows and columns so that the data can be viewed in the OLAPDataGrid control.

The following table describes the classes and interfaces that you use the define a query:

Class	Interface	Description
AsyncToken		The result of the `OLAPQuery.execute()` method.
OLAPCell	IOLAPCell	An area of the cube defined by a tuple.
OLAPMember	IOLAPMember	A member of a dimension.
OLAPQuery	IOLAPQuery	The query instances.
OLAPQueryAxis	IOLAPQueryAxis	An axis of the query.
OLAPResult	IOLAPResult	The query result.
OLAPSet		A set of members for an axis.
OLAPTuple	IOLAPTuple	A tuple containing one or more members.

Note: *Many of the examples in this topic use interfaces rather than classes. One of the reasons to use interfaces is that if you define customized versions of classes that implement these interfaces, you do not have to modify your code.*

Creating a query

The process of creating a query has the following steps:

1 Prepare the cube for a query. For more information, see "Preparing a cube for a query" on page 334.

2 Define an instance of the OLAPQuery class to represent the query. For more information, see "Creating a query axis" on page 335.

3 Define an instance of the OLAPQueryAxis class to represent the rows of the query.

4 Define an instance of the OLAPSet class to define the members that provide the row axis information. For more information, see "Writing a query for a simple OLAP cube" on page 338 and "Writing a query for a complex OLAP cube" on page 344.

5 Define an instance of the OLAPQueryAxis class to represent the columns of the query.

6 Define an instance of the OLAPSet class to define the members that provide the axis column information.

7 Optionally define an instance of the OLAPQueryAxis class to specify a slicer axis. For more information, see "Creating a slicer axis" on page 349

8 Define an instance of the OLAPSet class to define the members that provide the slicer axis information.

9 Execute the query on the cube by calling `OLAPCube.execute()`. For more information, see "Executing a query and returning the results to an OLAPDataGrid control" on page 336.

10 Pass the query results to the OLAPDataGrid control.

Preparing a cube for a query

Before you can run the first query on an OLAP cube, you must call the `OLAPCube.refresh()` method to initialize the cube from the input data. Upon completion of cube initialization, the OLAP cube dispatches the `complete` event to signal that the cube is ready for you to query.

You can use event handlers to initialize the cube and to invoke the query. The following example uses the application's `creationComplete` event to initialize the cube, and then uses the cube's `complete` event to execute the query:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    creationComplete="creationCompleteHandler();">

<mx:Script>
```

```
    <![CDATA[

        private function creationCompleteHandler():void {
            // You must initialize the cube before you
            // can execute a query on it.
            myMXMLCube.refresh();
        }

        // Event handler to execute the OLAP query
        // after the cube completes initialization.
        private function runQuery(event:CubeEvent):void {
            ...
        }
    ]]>
</mx:Script>

<mx:OLAPCube name="FlatSchemaCube"
    dataProvider="{flatData}"
    id="myMXMLCube"
    complete="runQuery(event);">
    ...
</mx:OLAPCube>
```

Creating a query axis

The following example shows an OLAPDataGrid control containing the results of a query for the sales of different products for different quarters:

Product	Quarter			
	Q1	Q2	Q3	Q4
ColdFusion	1485	300	785	430
Flex	420	1430	310	730
DreamWeaver	980	350	1990	590
Flash	650	1190	1090	2250

To create a query, you must define a row axis, a column axis, and optionally a slicer axis, where each axis is an instance of the IOLAPQueryAxis interface. The following table describes the different types of axes:

Axis	Description
Row	Defines the data that appears in each row of the OLAPDataGrid control. In the previous image, you define the row to show each product. This axis is required.
Column	Defines the data displayed in the column. In the previous image, you define the columns to show each year. This axis is required.
Slicer	Optionally defines a filter to reduce the size of the query results, often to reduce the dimensionality of the results from a dimension greater than two so that you can display the results in an OLAPDataGrid control. You also use a slicer access to return data aggregations for the nondefault measure. For more information on using a slicer axis, see "Creating a slicer axis" on page 349.

When you construct the OLAPQueryAxis instance, you use the `OLAPQuery.getAxis()` method to initialize the axis to configure it as either a row, column, or slicer axis, as the following example shows:

```
// Create an instance of OLAPQuery to represent the query.
var query:OLAPQuery = new OLAPQuery;
```

```
// Get the row axis from the query instance.
var rowQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.ROW_AXIS);
// Create an OLAPSet instance to configure the axis.
...

var colQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.COLUMN_AXIS);
...
```

Populating a query axis

You use the OLAPSet class to specify the members of the dimensions of the OLAP cube that populate a query axis. To add a member to an OLAPSet instance, you call the OLAPSet.addElement() or OLAPSet.addElements() method. The following table describes these methods:

OLAPSet method	Description
`addElement(e:IOLAPElement):void`	Adds a single element to the set. • If you pass an IOLAPHierarchy or IOLAPAttributeHierarchy instance to the method, it adds the default member of the hierarchy to the set. • If you pass an IOLAPLevel instance to the method, it adds all the members of the level to the set. • If you pass an IOLAPMember instance, it adds the member to the set.
`addElements(members:IList):void`	Adds multiple elements to the set.

For more information, see "Writing a query for a simple OLAP cube" on page 338 and "Writing a query for a complex OLAP cube" on page 344.

Executing a query and returning the results to an OLAPDataGrid control

OLAP cubes can be complex, so you do not want your application to pause while Flex calculates the results of an OLAP query. Therefore, when you execute a query, you also set up two callback functions that handle the results of the query. Flex then calls these functions when the query completes. This architecture lets the query run asynchronously so that your application can continue to execute during query processing.

You execute a query by calling the `OLAPCube.execute()` method, where the `OLAPCube.execute()` method returns an instance of the AsyncToken class. You use the AsyncToken class, along with the AsyncResponder class, to specify the two callback functions to handle the query results when execution completes.

In this example, the function `showResult()` handles the query results when the query succeeds, and the function `showFault()` handles any errors detected during query execution:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
        creationComplete="creationCompleteHandler();">

<mx:Script>
    <![CDATA[

        private function creationCompleteHandler():void {
            // You must initialize the cube before you
            // can execute a query on it.
            myMXMLCube.refresh();
        }

        // Create the OLAP query.
        private function getQuery(cube:IOLAPCube):IOLAPQuery {
            ...
        }
```

```
        // Event handler to execute the OLAP query
        // after the cube completes initialization.
        private function runQuery(event:CubeEvent):void {
            // Get cube.
            var cube:IOLAPCube = IOLAPCube(event.currentTarget);
            // Create a query instance.
            var query:IOLAPQuery = getQuery(cube);
            // Execute the query.
            var token:AsyncToken = cube.execute(query);
            // Set up handlers for the query results.
            token.addResponder(new AsyncResponder(showResult, showFault));
        }

        // Handle a query fault.
        private function showFault(error:ErrorMessage, token:Object):void {
            Alert.show(error.faultString);
        }

        // Handle a query success.
        private function showResult(result:Object, token:Object):void {
            if (!result) {
                Alert.show("No results from query.");
                return;
            }

            myOLAPDG.dataProvider= result as OLAPResult;
        }
    ]]>
</mx:Script>

<mx:OLAPCube name="FlatSchemaCube"
    dataProvider="{flatData}"
    id="myMXMLCube"
    complete="runQuery(event);">
    ...
</mx:OLAPCube>

<mx:OLAPDataGrid id="myOLAPDG" width="100%" height="100%" />
```

Using the default member in a query

Many of the queries shown so far have referenced only two dimensions of the OLAP cube, such as ProductDim and TimeDim. But what happens to the other dimensions when you omit them from the query?

All dimensions have a default member, either explicit or implicit. When you execute a query that does not reference a dimension, the query aggregates the data for that dimension using the default member. For information on defining the default member of a schema, see "Creating a default member in a schema" on page 332.

Specifying a default cell string for the OLAPDataGrid control

When you define your query, you might create a situation where a cell of the OLAPDataGrid does not contain a value. For example, when aggregating data by customer and product, you might have a customer that has never purchased Illustrator. For that customer and product combination, the corresponding cell of the OLAPDataGrid displays the String "NaN".

You can customize this String by setting the `OLAPDataGrid.defaultCellString`. For example, if you want to set the String to "No value", you would create an OLAPDataGrid control as shown here:

```
<mx:OLAPDataGrid id="myOLAPDG"
```

```
    defaultCellString="No value"
    width="100%" height="100%"/>
```

Writing a query for a simple OLAP cube

In a simple OLAP cube, all of the dimensions of the cube define a single level. For example, you have flat data that contains three fields:

```
data:Object = {
    product:"ColdFusion"
    quarter :"Q1"
    revenue: "100.00",
}
```

The data fields of each record can contain the following values:

- The product field can have the values: ColdFusion, Flex, Dreamweaver, and Illustrator.
- The quarter field can have the values: Q1, Q2, Q3, and Q4.

The OLAP schema for this data defines two dimensions, each with a single level, as the following code shows:

```
<mx:OLAPCube name="FlatSchemaCube"
    dataProvider="{flatData}"
    id="myMXMLCube"
    complete="runQuery(event);">

    <mx:OLAPDimension name="ProductDim">
        <mx:OLAPAttribute name="Product" dataField="product"/>
        <mx:OLAPHierarchy name="ProductHier" hasAll="true">
            <mx:OLAPLevel attributeName="Product"/>
        </mx:OLAPHierarchy>
    </mx:OLAPDimension>

    <mx:OLAPDimension name="QuarterDim">
        <mx:OLAPAttribute name="Quarter" dataField="quarter"/>
        <mx:OLAPHierarchy name="QuarterHier" hasAll="true">
            <mx:OLAPLevel attributeName="Quarter"/>
        </mx:OLAPHierarchy>
    </mx:OLAPDimension>

    <mx:OLAPMeasure name="Revenue"
        dataField="revenue"
        aggregator="SUM"/>
</mx:OLAPCube>
```

After you call `OLAPCube.refresh()` to initialize the cube, it has the following structure:

```
ProductDim                  // Dimension
    Product                     // Hierarchy
        (All)                       // Member
        ColdFusion                  // Member
        Flex                        // Member
        Dreamweaver                 // Member
        Illustrator                 // Member

    ProductHier                 // Hierarchy
        (All)                       // Level
            ColdFusion                  // Member
            Flex                        // Member
            Dreamweaver                 // Member
```

```
            Illustrator                 // Member

        Product                     // Level
            ColdFusion                  // Member
            Flex                        // Member
            Dreamweaver                 // Member
            Illustrator                 // Member

QuarterDim                  // Dimension
    Quarter                     // Hierarchy
        (All)                       // Member
        Q1                          // Member
        Q2                          // Member
        Q3                          // Member
        Q4                          // Member

    QuarterHier                 // Hierarchy
        (All)                       // Level
            Q1                          // Member
            Q2                          // Member
            Q3                          // Member
            Q4                          // Member

        Quarter                     // Level
            Q1                          // Member
            Q2                          // Member
            Q3                          // Member
            Q4                          // Member
```

Notice in this cube:

- The ProductDim dimension contains two hierarchies: Product and ProductHier. The cube creates a hierarchy for each level specified by the OLAPLevel class, and for each hierarchy specified by the OLAPHierarchy class. The same is true for the QuarterDim dimension; it also contains two hierarchies.
- The order of the members, meaning the values along each dimension, is based on the order in which the members appear in the flat data. In this example, the quarters appear in order of Q1, Q2, Q3, and Q4 because the flat data was sorted by quarter. However, if the data was not sorted by quarter, the members along the QuarterDim could appear in any order.
- Each dimension contains a single level; therefore the `(All)` level contains the same data as the first level in the hierarchy. For example, the `(All)` level of the ProdcutHier hierarchy contains the same data as the Product level. For a complex cube, the `(All)` level would contain additional information.

For an example of a complex cube, see "Writing a query for a complex OLAP cube" on page 344.

Defining a query for a simple cube

An OLAP query has the following basic form:

```
// Create an instance of OLAPQuery to represent the query.
var query:OLAPQuery = new OLAPQuery;

// Get the row axis from the query instance.
var rowQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.ROW_AXIS);
// Create an OLAPSet instance to configure the axis.
var productSet:OLAPSet = new OLAPSet;
// Use OLAPSet.addElements() or OLAPSet.addElement() to add members to the row axis.
productSet.addElements(...);
// Add the OLAPSet instance to the axis.
rowQueryAxis.addSet(productSet);
```

```
// Get the column axis from the query instance, and configure it
// to aggregate the columns by the Quarter dimension.
var colQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.COLUMN_AXIS);
var quarterSet:OLAPSet= new OLAPSet;
// Use OLAPSet.addElements() or OLAPSet.addElement() to add members to the column axis.
productSet.addElements(...);
colQueryAxis.addSet(quarterSet);
```

You use the following methods and properties to extract information from an OLAP cube based on the attributes of the dimension, and pass it to the `OLAPSet.addElements()` or `OLAPSet.addElement()` method:

Method and property	Description
`OLAPCube.findDimension(name:String):IOLAPDimension`	Returns a dimension of a schema as an instance of the OLAPDimension class.
`OLAPDimension.findAttribute(name:String):IOLAPAttributeHierarchy`	Returns an instance of the IOLAPAttributeHierarchy interface, which is an attribute hierarchy that includes an IList instance of all members of the attribute.
`OLAPDimension.findMember(name:String):IOLAPMember`	Returns an IOLAPMember instance that represents a member with the specified name within the dimension.
`IOLAPAttributeHierarchy.children`	Contains all members of the level as an IList instance, but does not include the (All) member.
`IOLAPAttributeHierarchy.members`	Contains all members of the level as an IList instance, including the (All) member.

The following table shows the information returned for combinations of these methods and properties:

Reference to method and property	Returns
`findDimension("ProductDim").findAttribute("Product").children`	ColdFusion, Flex, Dreamweaver, and Illustrator
`findDimension("ProductDim").findAttribute("Product").members`	(All), ColdFusion, Flex, Dreamweaver, and Illustrator
`findDimension("ProductDim").findMember("Flex")`	Flex
`findDimension("QuarterDim").findAttribute("Quarter").children`	Q1, Q2, Q3, and Q4
`findDimension("QuarterDim").findAttribute("Quarter").members`	(All), Q1, Q2, Q3, and Q4
`findDimension("QuarterDim").findMember("Q2")`	Q2

Rather than use the `findAttribute()` method, you can drill down through the hierarchy of the cube by calling the following methods:

Method	Description
`OLAPDimension.findHierarchy(name:String):IOLAPHierarchy`	Returns a hierarchy of a dimension as an instance of OLAPHierarchy. You can specify either an OLAPHierarchy or an OLAPLevel instance to this method.
`OLAPHierarchy.findLevel(name:String):IOLAPLevel`	Returns a level of a hierarchy as an instance of OLAPLevel.
`OLAPLevel.findMember(name:String):IList`	Returns an IList instance that contains all IOLAPMember instances that match the String argument. You can then use the `IList.getItemAt()` method to access the any element in the IList.

The following table shows the information returned for different combinations of these methods:

Reference to method	Returns
`findDimension("ProductDim").findhHierarchy("ProductHier").findLevel("Product").children`	ColdFusion, Flex, Dreamweaver, and Illustrator
`findDimension("ProductDim").findHierarchy("ProductHier").findLevel("Product").members`	(All), ColdFusion, Flex, Dreamweaver, and Illustrator
`IOLAPElement(cube.findDimension("ProductDim").findHierarchy("ProductHier").findLevel("Product").findMember("Flex").getItemAt(0))`	Flex. Cast the result to an instance of IOLAPElement because `getItemAt()` returns an Object.
`findDimension("QuarterDim").findHierarchy("QuarterHier").findLevel("Quarter").children`	Q1, Q2, Q3, and Q4
`findDimension("QuarterDim").findHierarchy("QuarterHier").findLevel("Quarter").members`	(All), Q1, Q2, Q3, and Q4
`IOLAPElement(cube.findDimension("QuarterDim").findHierarchy("QuarterHier").findLevel("Quarter").findMember("Q2").getItemAt(0))`	Q2. Cast the result to an instance of IOLAPElement because `getItemAt()` returns an Object.

In the case of a simple cube, the `OLAPDimension.findHierarchy()`, `OLAPHierarchy.findLevel()`, and `OLAPLevel.findMember()` methods do not provide you with additional functionality from the `OLAPDimension.findAttribute()` and `OLAPDimension.findMember()` methods. They are more commonly used with complex cubes that contain dimensions with multiple levels. For more information, see "Writing a query for a complex OLAP cube" on page 344.

Add all members to an axis

The most common type of query returns a data aggregation for all members of an attribute of a schema. For example, you define an OLAP cube using a schema that contains a ProductDim and a QuarterDim dimension, as shown in the section "Writing a query for a simple OLAP cube" on page 338. You then want to generate a query for all products for all quarters. The following query extracts this information:

```
private function getQuery(cube:IOLAPCube):IOLAPQuery {
    // Create an instance of OLAPQuery to represent the query.
    var query:OLAPQuery = new OLAPQuery;

    // Get the row axis from the query instance.
    var rowQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.ROW_AXIS);
    // Create an OLAPSet instance to configure the axis.
    var productSet:OLAPSet = new OLAPSet;
    productSet.addElements(
```

```
        cube.findDimension("ProductDim").findAttribute("Product").children);
    rowQueryAxis.addSet(productSet);

    var colQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.COLUMN_AXIS);
    var quarterSet:OLAPSet= new OLAPSet;
    quarterSet.addElements(
        cube.findDimension("QuarterDim").findAttribute("Quarter").children);
    colQueryAxis.addSet(quarterSet);

    return query;
}
```

In this example, the column axis uses the `findDimension()` and `findAttribute()` methods to drill down into the ProductDim and QuarterDim dimensions to extract sales data. For each axis, you use the `IOLAPAttributeHierarchy.children` property to populate it with the all members of the attribute, but do not include the `(All)` member.

This query uses the default measure to populate the query result, where the default measure is the first measure defined in the cube's schema. Therefore, you do have to specify the measure as part of the query. For information on explicitly specifying the measure in the query, see "Creating a query using a nondefault measure" on page 349.

Notice that in this example, you did not have to specify the hierarchy name, ProductHier and QuarterDim, to extract the member from the schema. It is unnecessary to specify the hierarchy name when you only want to extract data for a single level of a dimension.

However, you could rewrite this example to explicitly drill down through the cube by calling the methods `OLAPDimension.findHierarchy()` and `OLAPHierarchy.findLevel()`, as the following example shows:

```
private function getQuery(cube:IOLAPCube):IOLAPQuery {
    // Create an instance of OLAPQuery to represent the query.
    var query:OLAPQuery = new OLAPQuery;

    // Get the row axis from the query instance.
    var rowQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.ROW_AXIS);
    // Create an OLAPSet instance to configure the axis.
    var productSet:OLAPSet = new OLAPSet;
    productSet.addElements(cube.findDimension(
        "ProductDim").findHierarchy("ProductHier").findLevel("Product").members);
    rowQueryAxis.addSet(productSet);

    var colQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.COLUMN_AXIS);
    var quarterSet:OLAPSet= new OLAPSet;
    quarterSet.addElements(
        cube.findDimension("QuarterDim").findHierarchy("QuarterHier").
        findLevel("Quarter").members);
    colQueryAxis.addSet(quarterSet);

    return query;
}
```

In this example, you drill down through the dimension to access the Product and Quarter levels. You typically use this technique with complex cubes where you are interested in an explicit member of a dimension.

Add explicit members to a query

In the query shown in the previous section, you obtain sales data for all members of the Product and Quarter levels of the schema. But, what if you only want sales data for a single product, or for a single quarter? When you drill down into a dimension, you can specify the individual member of the dimension that you want to query.

For example, you want to create a query to aggregate quarterly sales data only for Flex, but not for any other products. You therefore use the `OLAPDimension.findMember()` method to specify the name of the member. This method takes a String containing the name of the member, and returns an IOLAPMember instance that defines the member, as the following example shows:

```
private function getQuery(cube:IOLAPCube):IOLAPQuery {
    // Create an instance of OLAPQuery to represent the query.
    var query:OLAPQuery = new OLAPQuery;

    // Get the row axis from the query instance.
    var rowQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.ROW_AXIS);
    // Create an OLAPSet instance to configure the axis.
    var productSet:OLAPSet = new OLAPSet;
    productSet.addElement(cube.findDimension("ProductDim").findMember("Flex"));
    rowQueryAxis.addSet(productSet);

    var colQueryAxis:IOLAPQueryAxis =
    query.getAxis(OLAPQuery.COLUMN_AXIS);
    var quarterSet:OLAPSet= new OLAPSet;
    quarterSet.addElements(
        cube.findDimension("QuarterDim").findAttribute("Quarter").children);
    colQueryAxis.addSet(quarterSet);

    return query;
}
```

Notice that in the previous example, you use the `OLAPSet.addElement()` method to add the Flex member to the OLAPSet instance, rather than the `OLAPSet.addElements()` method. This is because you are adding a single member to the axis, rather than multiple members.

The only issue with using the `OLAPDimension.findMember()` method is that it returns the first IOLAPMember instance in the cube that matches the String argument. If you think that you might have multiple instances of a member, you can rewrite this example to explicitly drill down through the cube by calling the `OLAPDimension.findHierarchy()`, `OLAPHierarchy.findLevel()`, and `OLAPLevel.findMember()` methods. This situation is more common with cubes that contain complex dimensions. For more information and examples, see "Writing a query for a complex OLAP cube" on page 344.

The `OLAPLevel.findMember()` method returns an IList instance that contains all IOLAPMember instances that match the String argument. You can then use the `IList.getItemAt()` method to access any element in the IList, as the following example shows:

```
private function getQuery(cube:IOLAPCube):IOLAPQuery {
    // Create an instance of OLAPQuery to represent the query.
    var query:OLAPQuery = new OLAPQuery;

    // Get the row axis from the query instance.
    var rowQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.ROW_AXIS);
    // Create an OLAPSet instance to configure the axis.
    var productSet:OLAPSet = new OLAPSet;
    // Get the first IOLAPElement instance in the IList instance.
    productSet.addElement(
        IOLAPElement(cube.findDimension("ProductDim").findHierarchy("ProductHier").
        findLevel("Product").findMember("Flex").getItemAt(0)));
    rowQueryAxis.addSet(productSet);

    var colQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.COLUMN_AXIS);
    var quarterSet:OLAPSet= new OLAPSet;
    quarterSet.addElement(IOLAPMember(cube.findDimension("TimeDim").
        findHierarchy("Month").findMember("January")));
    quarterSet.addElement(IOLAPMember(cube.findDimension("TimeDim").
```

```
        findHierarchy("Month").findMember("February")));
    colQueryAxis.addSet(quarterSet);

    return query;
}
```

Writing a query for a complex OLAP cube

In a complex cube, at least one dimension of the schema contains multiple levels. For example, you have flat data that contains three fields:

```
data:Object = {
    product:"ColdFusion"
    year :"2006"
    quarter :"Q1"
    revenue: "100.00",
}
```

The data fields of each record can contain the following values:

- The product field can have the values: ColdFusion, Flex, Dreamweaver, and Illustrator.
- The year field can have the values: 2006 and 2007.
- The quarter field can have the values: Q1, Q2, Q3, and Q4.

In this example, your schema defines two dimensions, with the TimeDim dimension containing levels for year and quarter, as the following code shows:

```
<mx:OLAPCube name="FlatSchemaCube"
    dataProvider="{flatData}"
    id="myMXMLCube"
    complete="runQuery(event);">

    <mx:OLAPDimension name="ProductDim">
        <mx:OLAPAttribute name="Product" dataField="product"/>
        <mx:OLAPHierarchy name="ProductHier" hasAll="true">
            <mx:OLAPLevel attributeName="Product"/>
        </mx:OLAPHierarchy>
    </mx:OLAPDimension>

    <mx:OLAPDimension name="TimeDim">
        <mx:OLAPAttribute name="Year" dataField="year"/>
        <mx:OLAPAttribute name="Quarter" dataField="quarter"/>
        <mx:OLAPHierarchy name="Time-PeriodHier" hasAll="true">
            <mx:OLAPLevel attributeName="Year"/>
            <mx:OLAPLevel attributeName="Quarter"/>
        </mx:OLAPHierarchy>
    </mx:OLAPDimension>

    <mx:OLAPMeasure name="Revenue"
        dataField="revenue"
        aggregator="SUM"/>
</mx:OLAPCube>
```

The cube has the following structure:

```
ProductDim                  // Dimension
    Product                     // Hierarchy
        (All)                       // Member
        ColdFusion                  // Member
        Flex                        // Member
```

```
        Dreamweaver                 // Member
        Illustrator                 // Member

    ProductHier                 // Hierarchy
        (All)                       // Level
            ColdFusion                  // Member
            Flex                        // Member
            Dreamweaver                 // Member
            Illustrator                 // Member

        Product                     // Level
            ColdFusion                  // Member
            Flex                        // Member
            Dreamweaver                 // Member
            Illustrator                 // Member

TimeDim                     // Dimension
    Year                        // Hierarchy
        (All)                       // Member
        2006                        // Member
        2007                        // Member

    Quarter                     // Hierarchy
        (All)                       // Member
        Q1                          // Member
        Q2                          // Member
        Q3                          // Member
        Q4                          // Member

    Time-PeriodHier             // Hierarchy
        (All)                       // Level
            2006
                Q1
                Q2
                Q3
                Q4
            2007
                Q1
                Q2
                Q3
                Q4

        Year                        // Level
            2006                        // Member
            2007                        // Member

        Quarter                     // Level
            Q1 (having 2006 as a parent) // Member
            Q1 (having 2007 as a parent) // Member
            Q2 (having 2006 as a parent) // Member
            Q2 (having 2007 as a parent) // Member
            Q3 (having 2006 as a parent) // Member
            Q3 (having 2007 as a parent) // Member
            Q4 (having 2006 as a parent) // Member
            Q4 (having 2007 as a parent) // Member
```

Notice in this cube:

• The information for the quarter is added as a hierarchy under the TimeDim dimension, and as a level under the Time-PeriodHier hierarchy.

- The Quarter level under the Time-PeriodHier hierarchy contains multiple entries for each quarter. In this example, there is an entry for each quarter for each year.
- The order of the members, meaning the values along each dimension, is based on the order in which the member values appear in the flat data.

The reason for the multiple entries for a single quarter in the Quarter level under the Time-PeriodHier hierarchy is that a cube has to be able to return results for each quarter for each year. For example, this structure lets you write a query to return data aggregations for all quarters for all years, for all quarters for 2006, or for all quarters for 2007.

Since the ProductDim contains a single level, you can write queries for it in the same way as you did for a simple cube. See "Writing a query for a simple OLAP cube" on page 338 for more information.

The following table shows the information returned for different ways to access the TimeDim by calling the `findAttribute()` method:

Reference to method and property	Returns
`findDimension("TimeDim").findAttribute("Year").children`	2006, 2007
`findDimension("TimeDim").findAttribute("Year").members`	(All), 2006, 2007
`findDimension("TimeDim").findMember("2006")`	2006
`findDimension("TimeDim").findAttribute("Quarter").children`	Q1, Q2, Q3, and Q4
`findDimension("TimeDim").findAttribute("Quarter").members`	(All), Q1, Q2, Q3, and Q4
`findDimension("TimeDim").findMember("Q2")`	Q2 having 2006 as a parent. This is the first instance of Q2 in the cube.

The following table shows the information returned for different ways to use the `OLAPDimension.findHierarchy()`, `OLAPHierarchy.findLevel()`, and `OLAPLevel.findMember()` methods:

Reference to method and property	Returns
`findDimension("TimeDim").findHierarchy("Time-PeriodHier").findLevel("Year").children`	2006, 2007
`findDimension("TimeDim").findHierarchy("Time-PeriodHier").findLevel("Year").members`	(All), 2006, 2007
`IOLAPElement(cube.findDimension("TimeDim").findHierarchy("Time-PeriodHier").findLevel("Year").findMember("2006").getItemAt(0))`	2006
`findDimension("QuarterDim").findHierarchy("QuarterHier").findLevel("Quarter").children`	Q1, Q2, Q3, and Q4
`findDimension("QuarterDim").findHierarchy("QuarterHier").findLevel("Quarter").members`	(All), Q1, Q2, Q3, and Q4
`findDimension("TimeDim").findHierarchy("Time-PeriodHier").findLevel("Quarter").findMember("Q1")`	All Q1 members
`IOLAPElement(cube.findDimension("QuarterDim").findHierarchy("QuarterHier").findLevel("Quarter").findMember("Q2").getItemAt(0))`	Q2 having 2006 as a parent. Cast the result to an instance of IOLAPElement because `getItemAt()` returns an Object.
`IOLAPElement(cube.findDimension("QuarterDim").findHierarchy("QuarterHier").findLevel("Quarter").findMember("Q2").getItemAt(1))`	Q2 having 2007 as a parent. Cast the result to an instance of IOLAPElement because `getItemAt()` returns an Object.

You could add a Month level to the TimeDim, as the following definition of TimeDim shows:

```
<mx:OLAPDimension name="TimeDim">
```

```
    <mx:OLAPAttribute name="Year" dataField="year"/>
    <mx:OLAPAttribute name="Quarter" dataField="quarter"/>
    <mx:OLAPAttribute name="Month" dataField="month"/>
    <mx:OLAPHierarchy name="Time-PeriodHier" hasAll="true">
        <mx:OLAPLevel attributeName="Year"/>
        <mx:OLAPLevel attributeName="Quarter"/>
        <mx:OLAPLevel attributeName="Month"/>
    </mx:OLAPHierarchy>
</mx:OLAPDimension>
```

For this example, the structure of the TimeDim dimension is:

```
TimeDim                     // Dimension
    Year                        // Hierarchy
        (All)                       // Member
        2006                        // Member
        2007                        // Member

    Quarter                     // Hierarchy
        (All)                       // Member
        Q1                          // Member
        Q2                          // Member
        Q3                          // Member
        Q4                          // Member

    Month                       // Hierarchy
        (All)                       // Member
        Jan                         // Member
        Feb                         // Member
        ...                         // Adiitional members for each month

    Time-PeriodHier             // Hierarchy
        (All)                       // Level
            2006
                Q1
                    Jan
                    Feb
                    Mar
                ...
            2007
                Q1
                    Jan
                    Feb
                    Mar
                ...

        Year                        // Level
            2006                        // Member
            2007                        // Member

        Quarter                     // Level
            Q1 (having 2006 as a parent) // Member
            Q1 (having 2007 as a parent) // Member
            Q2 (having 2006 as a parent) // Member
            Q2 (having 2007 as a parent) // Member
            Q3 (having 2006 as a parent) // Member
            Q3 (having 2007 as a parent) // Member
            Q4 (having 2006 as a parent) // Member
            Q4 (having 2007 as a parent) // Member

        Month                       // Level
            Jan (having Q1 having 2006 as a parent) // Member
```

```
Feb (having Q1 having 2006 as a parent) // Member
Jan (having Q1 having 2007 as a parent) // Member
Feb (having Q1 having 2007 as a parent) // Member
...                                     // Additional members for each possible
                                        // combination of month, quarter,
                                        // and year
```

Notice in this cube:

- The Month level contains a value for each month, and for each quarter for each year. Therefore, there should be two entries for January; one for Q1 of 2006 and one for Q1 of 2007.

Creating a multidimensional axis in an OLAPDataGrid control

The OLAPDataGrid control displays information along two axes: row and column. However, limiting yourself to writing queries that return only two dimensions can restrict your ability to examine your data.

To allow you greater flexibility in displaying query results, the OLAPDataGrid control supports hierarchical display along its axis. The following image shows quarterly sales information, grouped by year, displayed in the columns of the OLAPDataGrid control.

Product	Year	Quarter						
	2006				2007			
	Q1	Q2	Q3	Q4	Q1	Q2	Q3	Q4
ColdFusion	770	125	375	215	715	175	410	215
Flex	210	760	210	430	210	670	100	300
DreamWeaver	470	175	840	320	510	175	1150	270
Flash	280	570	670	1010	370	620	420	1240

To create a multidimensional axis in the OLAPDataGrid control, you create two or more OLAPSet instances for the axis, and then combine the OLAPSet instances by doing a crossjoin or a union, as the following table describes:

Combination type	OLAPSet method	Description
Crossjoin	`crossJoin()`	Creates a crossjoin of two OLAPSet instances, where a crossjoin contains all possible combinations of the two sets. A crossjoin is also called a cross product of the members of the two different sets.
Union	`union()`	Creates a union of two OLAPSet instances.

In the following example, you create an OLAPSet instance for the year, and then crossjoin that set with the OLAPSet instance for quarter to create the OLAPDataGrid control shown in the previous image:

```
private function getQuery(cube:IOLAPCube):IOLAPQuery {
    var query:OLAPQuery = new OLAPQuery;

    var rowQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.ROW_AXIS);
    var productSet:OLAPSet = new OLAPSet;
    productSet.addElements(
        cube.findDimension("ProductDim").findAttribute("Product").children);
    rowQueryAxis.addSet(productSet);
```

```
    var colQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.COLUMN_AXIS);
    var yearSet:OLAPSet= new OLAPSet;
    yearSet.addElements(
        cube.findDimension("TimeDim").findAttribute("Year").children);
    var quarterSet:OLAPSet= new OLAPSet;
    quarterSet.addElements(
        cube.findDimension("TimeDim").findAttribute("Quarter").children);
    colQueryAxis.addSet(yearSet.crossJoin(quarterSet));

    return query;
}
```

Creating a slicer axis

An OLAP query can have three axes: row, column, and slicer. A slicer axis lets you reduce the size of the query results, often to reduce the dimensionality of the results from a dimension greater than two so that you can display the results. Another common use of a slicer axis is to aggregate data on a measure other than the default measure.

Creating a query using a nondefault measure

The following schema defines two measures for the points in the cube: Revenue and Cost. The first measure defined in the schema is the default measure, and it is the data that is aggregated by a query if you do not explicitly specify the measure.

```
<mx:OLAPCube name="FlatSchemaCube"
    dataProvider="{flatData}"
    id="myMXMLCube"
    complete="runQuery(event);">

    <mx:OLAPDimension name="ProductDim">
        <mx:OLAPAttribute name="Product" dataField="product"/>
        <mx:OLAPHierarchy name="ProductHier" hasAll="true">
            <mx:OLAPLevel attributeName="Product"/>
        </mx:OLAPHierarchy>
    </mx:OLAPDimension>

    <mx:OLAPDimension name="QuarterDim">
        <mx:OLAPAttribute name="Quarter" dataField="quarter"/>
        <mx:OLAPHierarchy name="QuarterHier" hasAll="true">
            <mx:OLAPLevel attributeName="Quarter"/>
        </mx:OLAPHierarchy>
    </mx:OLAPDimension>

    <mx:OLAPMeasure name="Revenue"
        dataField="revenue" aggregator="MAX"/>
    <mx:OLAPMeasure name="Cost"
        dataField="cost" aggregator="MIN"/>
</mx:OLAPCube>
```

To aggregate by the Cost measure, you create a slicer axis to explicitly specify the measure for the query, as the following example shows:

```
private function getQuery(cube:IOLAPCube):IOLAPQuery {
    // Create an instance of OLAPQuery to represent the query.
    var query:OLAPQuery = new OLAPQuery;

    // Get the row axis from the query instance.
    var rowQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.ROW_AXIS);
    // Create an OLAPSet instance to configure the axis.
    var productSet:OLAPSet = new OLAPSet;
```

```
    productSet.addElements(
        cube.findDimension("ProductDim").findAttribute("Product").children);
    rowQueryAxis.addSet(productSet);

    var colQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.COLUMN_AXIS);
    var quarterSet:OLAPSet= new OLAPSet;
    quarterSet.addElements(
        cube.findDimension("TimeDim").findAttribute("Month").children);
    colQueryAxis.addSet(quarterSet);

    // Create the slicer axis.
    var slicerQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.SLICER_AXIS);
    // Create an OLAPSet instance to configure the axis.
    var costSet:OLAPSet= new OLAPSet;
    // Use OLAPDimension.findMember() to add the Cost measure.
    costSet.addElement(cube.findDimension("Measures").findMember("Cost"));
    slicerQueryAxis.addSet(costSet);

    return query;
}
```

In this example, you use the keyword `Measures` to identify the measure dimension, and then use the `OLAPDimension.findMember()` method to access the Cost measure.

Using a slicer axis to reduce the dimensionality of the query result

The OLAPDataGrid control displays information in two dimensions along its row and column axis. However, to display product sales by region and by month, you require a three-dimensional display: one dimension each for product, region, and month.

For example, your data has the following format:

```
data:Object = {
    customer:"IBM",
    country:"US",
    state:"MA",
    region:"NewEngland",
    product:"ColdFusion",
    year:2005,
    quarter:"Q1"
    month:"January",
    revenue: 12,575.00,
    cost: 500
}
```

The examples use the following OLAP schema to represent this data in an OLAP cube:

```
 <mx:OLAPCube name="FlatSchemaCube"
        dataProvider="{flatData}"
        id="myMXMLCube"
        complete="runQuery(event);">

        <mx:OLAPDimension name="CustomerDim">
            <mx:OLAPAttribute name="Customer" dataField="customer"/>
            <mx:OLAPHierarchy name="CustomerHier"
                hasAll="true">
                <mx:OLAPLevel attributeName="Customer"/>
            </mx:OLAPHierarchy>
        </mx:OLAPDimension>

        <mx:OLAPDimension name="ProductDim">
            <mx:OLAPAttribute name="Product" dataField="product"/>
```

```
            <mx:OLAPHierarchy name="ProductHier"
                hasAll="true">
                <mx:OLAPLevel attributeName="Product"/>
            </mx:OLAPHierarchy>
        </mx:OLAPDimension>

        <mx:OLAPDimension name="TimeDim">
            <mx:OLAPAttribute name="Year" dataField="year"/>
            <mx:OLAPAttribute name="Quarter" dataField="quarter"/>
            <mx:OLAPAttribute name="Month" dataField="month"/>
            <mx:OLAPHierarchy name="Time-Period"
                hasAll="true">
                <mx:OLAPLevel attributeName="Year"/>
                <mx:OLAPLevel attributeName="Quarter"/>
                <mx:OLAPLevel attributeName="Month"/>
            </mx:OLAPHierarchy>
        </mx:OLAPDimension>

        <mx:OLAPDimension name="GeographyDim">
            <mx:OLAPAttribute name="Country" dataField="country"/>
            <mx:OLAPAttribute name="Region" dataField="region"/>
            <mx:OLAPAttribute name="State" dataField="state"/>
            <mx:OLAPHierarchy name="Country-Region-State"
                hasAll="true">
                <mx:OLAPLevel attributeName="Country"/>
                <mx:OLAPLevel attributeName="Region"/>
                <mx:OLAPLevel attributeName="State"/>
            </mx:OLAPHierarchy>
        </mx:OLAPDimension>

        <mx:OLAPMeasure name="Revenue"
            dataField="revenue"
            aggregator="SUM"/>
        <mx:OLAPMeasure name="Cost"
         dataField="cost"
         aggregator="SUM"/>
    </mx:OLAPCube>
```

You can use a slicer axis to filter the results of a query for display in two dimensions. In this example, you set the row axis to the region and the column axis to the month. You then define a slicer axis to specify the product as Flex so that you end up with a two-dimensional table of sales by region and month for Flex.

```
private function getQuery(cube:IOLAPCube):IOLAPQuery {
    // Create an instance of OLAPQuery to represent the query.
    var query:OLAPQuery = new OLAPQuery;

    // Get the row axis from the query instance.
    var rowQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.ROW_AXIS);
    // Create an OLAPSet instance to configure the axis.
    var productSet:OLAPSet = new OLAPSet;
    productSet.addElements(
        cube.findDimension("GeographyDim").findAttribute("Region").children);
    rowQueryAxis.addSet(productSet);

    var colQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.COLUMN_AXIS);
    var quarterSet:OLAPSet= new OLAPSet;
    quarterSet.addElements(
        cube.findDimension("TimeDim").findAttribute("Month").children);
    colQueryAxis.addSet(quarterSet);

    // Create the slicer axis.
    var slicerQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.SLICER_AXIS);
```

```
    // Create an OLAPSet instance to configure the axis.
    var flexSet:OLAPSet= new OLAPSet;
    flexSet.addElement(
        IOLAPElement(cube.findDimension("ProductDim").findHierarchy("ProductHier").
        findLevel("Product").findMember("Flex").getItemAt(0)));
    slicerQueryAxis.addSet(flexSet);

    return query;
}
```

You can specify more than one member for the slicer access. For example, if your cube contains information on different product such as Flex and Adobe® Flash®, you could create the two slicer axes as the following example shows:

```
// Create the slicer axis.
var slicerQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.SLICER_AXIS);

// Create an OLAPSet instance to configure the axis.
var sliceSet:OLAPSet= new OLAPSet;
sliceSet.addElement(IOLAPElement(cube.findDimension("ProductDim").
    findHierarchy("ProductHier").findLevel("Product").findMember("Flex").getItemAt(0)));

sliceSet.addElement(IOLAPElement(cube.findDimension("ProductDim").
    findHierarchy("ProductHier").findLevel("Product").findMember("Flash").getItemAt(0)));

slicerQueryAxis.addSet(sliceSet);
```

In this example, your query result would show sales of Flex and Flash for each region and month.

Configuring the display of an OLAPDataGrid

To control the display of an OLAPDataGrid control, you can apply styles to the cells of the control by using a callback function, or use item renderers to control the display.

Styling cells of the OLAPDataGrid control by using a callback function

To control the styling of a cell by using a callback function, use the `OLAPDataGrid.styleFunction` property to specify the callback function. The callback function must have the following signature:

```
function_name(row:IOLAPAxisPosition, column:IOLAPAxisPosition, value:Number):Object
```

where `row` is the IOLAPAxisPosition associated with this cell on the row axis, `column` is the IOLAPAxisPosition associated with this cell on the column axis, and `value` is the cell value.

The function returns an Object that contains one or more *styleName:value* pairs to specify a style setting, or null. The *styleName* field contains the name of a style property, such as `color`, and the *value* field contains the value for the style property, such as 0x00FF00. For example, you could return two styles using the following syntax:

```
{color:0xFF0000, fontWeight:"bold"}
```

The OLAPDataGrid control invokes the callback function when it updates its display, such as when the control is first drawn on application start up, or when you call the `invalidateList()` method.

The following example modifies the example shown in the section "Example using the OLAPDataGrid control" on page 327 to add a callback function to display all cells with a value greater than 1000 in green:

```
<mx:Script>
    <![CDATA[
        ...
```

```
        // Callback function that hightlights in green
        // all cells with a value greater than or equal to 1000.
        public function myStyleFunc(row:IOLAPAxisPosition, column:IOLAPAxisPosition,
            value:Number):Object
        {
            if (value >= 1000)
            return {color:0x00FF00};

            // Return null if value is less than 1000.
            return null;
        }
    ]]>
</mx:Script>

...

<mx:OLAPDataGrid id="myOLAPDG"
    width="100%" height="100%"
    styleFunction="myStyleFunc"/>
```

Using an item renderer with the OLAPDataGrid control

You customize the appearance and behavior of cells in an OLAPDataGrid control by creating custom item renderers. For an introduction to item renderers and item editors, see "Using Item Renderers and Item Editors" on page 779 in *Adobe Flex 3 Developer Guide*.

To use an item renderer with the OLAPDataGrid control, you assign the item renderer to the OLAPDataGrid control itself, not to a specific column, by using the `OLAPDataGrid.itemRendererProviders` property. The `itemRendererProviders` property contains an Array of OLAPDataGridItemRendererProvider instances, where each OLAPDataGridItemRendererProvider instance defines the characteristics for a single item renderer.

You can use the `OLAPDataGridRendererProvider.renderer` property to assign an item renderer to the OLAPDataGrid control, much in the way that you can by using the `AdvancedDataGrid.rendererProviders` property, or can use the `OLAPDataGridRendererProvider.formatter` property to assign a formatter class. The following example modifies the example shown in the section "Example using the OLAPDataGrid control" on page 327 to add an item renderer that applies the CurrencyFormatter formatter to each cell in the control:

```
...
<mx:CurrencyFormatter id="usdFormatter" precision="2"
    currencySymbol="$" decimalSeparatorFrom="."
    decimalSeparatorTo="." useNegativeSign="true"
    useThousandsSeparator="true" alignSymbol="left"/>

...

<mx:OLAPDataGrid id="myOLAPDG"
    width="100%" height="100%">

    <mx:itemRendererProviders>
        <mx:OLAPDataGridItemRendererProvider
            uniqueName="[QuarterDim].[Quarter]"
            type="{OLAPDataGrid.OLAP_HIERARCHY}"
            formatter="{usdFormatter}"/>
    </mx:itemRendererProviders>
</mx:OLAPDataGrid>
```

To assign the item renderer, use the `uniqueName` and `type` properties of the OLAPDataGridItemRendererProvider class. The `uniqueName` property specifies the elements of the query, and therefore the corresponding cells of the OLAPDataGrid control, that you modify by using the item renderer. The `type` properly specifies the element type, such as dimension, hierarchy, or level of the element specified by the `uniqueName` property.

The function that defines the query for this example is shown below:

```
// Create the OLAP query.
private function getQuery(cube:IOLAPCube):IOLAPQuery {
    // Create an instance of OLAPQuery to represent the query.
    var query:OLAPQuery = new OLAPQuery;

    // Get the row axis from the query instance.
    var rowQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.ROW_AXIS);
    // Create an OLAPSet instance to configure the axis.
    var productSet:OLAPSet = new OLAPSet;
    // Add the Product to the row to aggregate data
    // by the Product dimension.
    productSet.addElements(
        cube.findDimension("ProductDim").findAttribute("Product").children);
    // Add the OLAPSet instance to the axis.
    rowQueryAxis.addSet(productSet);

    // Get the column axis from the query instance, and configure it
    // to aggregate the columns by the Quarter dimension.
    var colQueryAxis:IOLAPQueryAxis = query.getAxis(OLAPQuery.COLUMN_AXIS);
    var quarterSet:OLAPSet= new OLAPSet;
    quarterSet.addElements(
        cube.findDimension("QuarterDim").findAttribute("Quarter").children);
    colQueryAxis.addSet(quarterSet);

    return query;
}
```

Notice that the axis for the QuarterDim creates a query for the Quarter hierarchy of the QuarterDim dimension. Therefore, you set the `uniqueName` property to `[QuarterDim].[Quarter]`, and set the type property to `OLAPDataGrid.OLAP_HIERARCHY`. For more information on the structure of the `OLAP` code for this example, see "Writing a query for a simple OLAP cube" on page 338.

You can modify this example to drill down through the cube by specifying the query as shown below for the QuarterDim:

```
quarterSet.addElements(
    cube.findDimension("QuarterDim").findHierarchy("QuarterHier").
    findLevel("Quarter").members);
```

You therefore modify the `uniqueName` and `type` properties as shown below:

```
<mx:OLAPDataGrid id="myOLAPDG"
    width="100%" height="100%">

    <mx:itemRendererProviders>
        <mx:OLAPDataGridItemRendererProvider
            uniqueName="[QuarterDim].[QuarterHier].[Quarter]"
            type="{OLAPDataGrid.OLAP_LEVEL}"
            formatter="{usdFormatter}"/>
    </mx:itemRendererProviders>
</mx:OLAPDataGrid>
```

Notice in this example that the query for the QuarterDim specifies the dimension, hierarchy, and level of the OLAP cube. Therefore, you modify the `uniqueName` property to match this structure, and set the `type` property to `OLAPDataGrid.OLAP_LEVEL`.

Resolving item renderer conflicts

Each cell in an OLAPDataGrid control is a result of an intersection between the members along a row and the members along a column of the control. However, when you assign an item renderer to an OLAPDataGrid control, you only specify the `uniqueName` and `type` properties for one of the dimensions, either row or column. Therefore, you can create a situation where two different item renderers are assigned to the same cell of the control.

In case of a conflict between two or more item renderers, the OLAPDataGrid control applies the item renderer based on the following priorities:

1. `type = OLAPDataGrid.OLAP_MEMBER`
2. `type = OLAPDataGrid.OLAP_LEVEL`
3. `type = OLAPDataGrid.OLAP_HIERARCHY`
4. `type = OLAPDataGrid.OLAP_DIMENSION`

Therefore, if an item renderer with a `type` value of `OLAPDataGrid.OLAP_LEVEL` and an item renderer with a `type` value of `OLAPDataGrid.OLAP_HIERARCHY` are applied to the same cell, the OLAPDataGrid control applies the item renderer with a `type` value of `OLAPDataGrid.OLAP_LEVEL`.

If two item renderers have the same value for the `type` property, the OLAPDataGrid control determines which renderer more closely matches the item, and uses it.

Using an item renderer to apply styles to an OLAPDataGrid control

You can use an item renderer to apply styles to specific cells in the OLAPDataGrid control, as the following example shows:

```
<mx:Style>
    .cellStyle
    {
        color:#ff0000;
        fontWeight:"bold"
    }
</mx:Style>

    ...

<mx:OLAPDataGrid id="myOLAPDG"
    width="100%" height="100%">

    <mx:itemRendererProviders>
        <mx:OLAPDataGridItemRendererProvider
            uniqueName="[QuarterDim].[Quarter]"
            type="{OLAPDataGrid.OLAP_HIERARCHY}"
            formatter="{usdFormatter}"
            styleName="cellStyle"/>
    </mx:itemRendererProviders>
</mx:OLAPDataGrid>
```

In this example, you create a style definition, then apply it to the cells of the OLAPDataGrid control so that it applies to the same cells as the CurrencyFormatter formatter.

Chapter 8: Creating Applications for Testing

You can create applications and components that can be tested with automated testing tools such as Mercury QuickTest Professional™ (QTP). The information in this topic is intended for Adobe® Flex™ developers who write applications that are tested by Quality Control (QC) professionals who use these testing tools. For information on installing and running the Flex plug-in with QTP, QC professionals should see *Testing Adobe Flex Applications with Mercury QuickTest Professional*.

Full support for the Flex automation features is included in Adobe® Flex® Builder™ Professional. Adobe Flex Builder Standard allows only limited use of this feature.

Topics

About automating applications with Flex 356
Tasks and techniques for testable applications overview 357
Compiling applications for testing 358
Creating testable applications 360
Understanding the automation framework 362
Instrumenting events 365
Instrumenting custom components 367
Instrumenting composite components 373
Example: Instrumenting the RandomWalk custom component for QTP 375

About automating applications with Flex

The Flex automation feature provides developers with the ability to create Flex applications that use the automation APIs. You can use these APIs to create automation agents or to ensure that your applications are ready for testing. In addition, the Flex automation feature includes support for the QTP automation tool.

The Flex automation feature includes the following:

- Automation libraries — The automation.swc, automation_dmv.swc, automation_agent.swc libraries are the implementations of the delegates for the Flex framework components. The automation_agent.swc file and its associated resource bundle are the generic agent mechanism. An agent such as QTP builds on top of these libraries. The automation_dmv.swc library provides the delegates for the Flex charting and AdvancedDataGrid classes.
- QTP files — The QTP files let QTP and Flex applications communicate directly. You can only use these files if you also have QTP. These files include the QTP plug-in, a QTP demonstration video, a QTP-specific environment XML file, and the QTP-specific library, qtp.swc. For more information, see *Testing Adobe Flex Applications with Mercury QuickTest Professional*.
- Samples — Sample applications include the CustomAgent, AutoQuick, RandomWalk, and FlexStore applications. The CustomAgent sample shows a simple agent that records metrics information for Flex applications. The AutoQuick example includes a component that lets you record and play back interactions with a Flex application. Testing-enabled versions of the FlexStore and RandomWalk applications are also available.

The following table shows the locations of the automation sample applications:

Automation-enabled application	Location
CustomAgent	http://www.adobe.com/go/flex_automation_agent_apps
AutoQuick	http://www.adobe.com/go/flex_automation_agent_apps
RandomWalk	http://www.adobe.com/go/flex_automation_randomwalk_apps
FlexStore	http://www.adobe.com/go/flex_flexstore_automation

When working with the automation APIs, you should understand the following terms:

- *automation agent* (or, simply, *agent*) — An agent facilitates communication between a Flex application and an automation tool. The Flex Automation Package includes a plugin that acts as an agent between your Flex applications and the QTP testing tool.
- *automation tool* — Automation tools are applications that use the Flex application data that is derived through the agent. These tools include QTP, Omniture, and Segue. In some cases.

The following illustration shows the relationship between a Flex application, an agent, and an automation tool.

Tasks and techniques for testable applications overview

Flex developers should review the information about tasks and techniques for creating testable applications, and then update their Flex applications accordingly. QC testing professionals who use QTP should use the documentation provided in the separate book, *Testing Adobe Flex Applications with Mercury QuickTest Professional*. That document is available for download with the Flex plug-in for QTP.

Use the following general steps to create a testable application:

1 Review the guidelines for creating testable applications. For more information, see "Creating testable applications" on page 360.

2 Prepare the application to use automation at run time or build your own testable application.

- To use automation at run time, you use a wrapper SWF file that is compiled with the automation libraries. This wrapper SWF file uses the SWFLoader to load your application SWF file that you plan to test only at run time. The HTML template and the MXML source code for the wrapper SWF file are included for you in the *flex_builder_install_dir*/sdks/3.0.0/templates/automation-runtimeloading-files directory. For more information, see "Use run-time loading" on page 359.

- To build a testable application, you include automation libraries at compile time. Compile the application's SWF file with the automation.swc and automation_agent.swc files specified in the compiler's `include-libraries` option. If your application uses charts or the AdvancedDataGrid classes, you must also add the automation_dmv.swc file. If you are using the QTP plug-in, add the qtp.swc file. For information on the compilation process, see "Compiling applications for testing" on page 358.

  If you build your own testable application, you must also create an HTML wrapper with proper object naming. If you are using Flex Builder, you can generate a wrapper automatically. If you are using the SDK, you can use the wrapper templates in the *flex_sdk*/templates directory to create a wrapper for your application. When using a wrapper, the value of the `id` attributes can not contain any periods or hyphens.

3 Prepare customized components for testing. If you have custom components that extend UIComponent, make them testable. For more information, see "Instrumenting custom components" on page 367.

4 Deploy the application's assets to a web server. Assets can include the SWF file; HTML wrapper and related files; external assets such as theme files, graphics, and video files; module SWF files; resource modules; CSS SWF files; and run-time shared libraries (RSLs). For information about what files to deploy with your application, see "Deployment checklist" on page 304 in *Building and Deploying Adobe Flex 3 Applications*.

Compiling applications for testing

If you do not use run-time testing, you must compile applications that you plan to test with the testing libraries. The functional testing classes are embedded in the application at compile time, and the application typically has no external dependencies for automated testing at run time.

For information about run-time testing, see "Use run-time loading" on page 359.

When you embed functional testing classes in your application SWF file at compile time, you increase the size of the SWF file. If the size of the SWF file is not important, you can use the same SWF file for functional testing and deployment. If the size of the SWF file is important, you typically generate two SWF files: one with functional testing classes embedded and one without.

To compile the Flex application for testing, you must reference the automation.swc and automation_agent.swc files in your `include-libraries` compiler option. If your application uses charts or the AdvancedDataGrid classes, you must also add the automation_dmv.swc file. You might also be required to add automation tool-specific SWC files; for example, for QTP, you must also add the qtp.swc file to your application's library path. If you are using the AutoQuick example, you must add the AutoQuick.swc file.

By default, Flex Builder includes the automation SWC files in its library path, but you must add these libraries by using the `include-libraries` compiler option.

When you create the final release version of your Flex application, you recompile the application without the references to these SWC files. For more information about using the automation SWC files, see the Flex Automation Release Notes.

If you do not deploy your application to a server, but instead request it by using the file protocol or run it from within Adobe Flex Builder, you must put the SWF file into the local-trusted sandbox. This requires configuration information that is separate from the SWF file and the wrapper. For more information that is specific to QTP, see *Testing Adobe Flex Applications with Mercury QuickTest Professional*.

To include the SWC files, you can add them to the compiler's configuration file or as a command-line option. For the SDK, the configuration file is located at *sdk_install_dir*/frameworks/flex-config.xml. To add the automation.swc and automation_agent.swc libraries, add the following lines to the configuration file:

```
<include-libraries>
    ...
```

```
    <library>/libs/automation.swc</library>
    <library>/libs/automation_agent.swc</library>
</include-libraries>
```

You must uncomment the `include-libraries` code block. By default it is commented out in the configuration file.

You can also specify the location of the SWC files when you use the command-line compiler with the `include-libraries` compiler option. The following example adds automation.swc and automation_agent.swc files to the application:

```
mxmlc -include-libraries+=../frameworks/libs/automation.swc;
    ../frameworks/libs/automation_agent.swc MyApp.mxml
```

Explicitly setting the `include-libraries` option on the command line overwrites, rather than appends, any existing libraries that you include in the configuration file. As a result, if you add the automation.swc and automation_agent.swc files by using the `include-libraries` option on the command line, ensure that you use the += operator. This appends rather than overwrites the existing libraries that are included.

To add automated testing support to a Flex Builder project, you also add the SWC files to the `include-libraries` compiler option.

Add SWC files to Flex Builder projects

1. In Flex Builder, select your Flex project in the Navigator.
2. Select Project > Properties. The Properties dialog box appears.
3. Select Flex Compiler in the tree to the left. The Flex Compiler properties panel appears.
4. In the "Additional compiler arguments" field, enter the following command:

   ```
   -include-libraries "sdks\3.0.0\frameworks\libs\automation.swc"
   "sdks\3.0.0\frameworks\libs\automation_agent.swc"
   ```

   In Flex Builder, the entries in the `include-libraries` compiler option are relative to the Flex Builder installation directory; the default location of this directory on Windows is C:\Program Files\Adobe\Flex Builder 3.

   If your application uses charts or the AdvancedDataGrid classes, you must also add the automation_dmv.swc file.
5. Click OK to save your changes.

Use run-time loading

You use the run-time testing files rather than building your applications with testing libraries. This lets you test SWF files that are compiled without automated testing support. To do this, you use a SWF file that *does* include the automated testing libraries. In that SWF file, the SWFLoader class loads your application's SWF file which does not include the testing libraries. The result is that you can test the target SWF file in a testing tool such as QTP, even though the application SWF file was not compiled with automated testing support.

Flex Builder includes a wrapper SWF file and an HTML template that supports run-time loading. The source MXML file for the wrapper SWF file is also included. The following files are located in the *flex_builder_install_dir*/sdks/3.0.0/templates/automation-runtimeloading-files directory:

- RunTimeLoading.html — The HTML template that loads the run-time loader SWF file. This template includes code that converts the `automationswfurl` query string parameter to a `flashVars` variable that it passes to the application. You use this query string parameter to specify the name of the application you want to test.
- runtimeloading.mxml — The source code for the runtimeloading.swf file that you compile. The SWF file acts as a wrapper for your application. This SWF file includes the testing libraries so that you do not have to compile them into your application SWF file.

1 Compile the runtimeloading.swf file. You can use the batch file in the *flex_builder_install_dir*/sdks/3.0.0/templates/automation-runtimeloading-files directory. Execute this batch file from the sdks/3.0.0/frameworks directory. This batch file ensures that your runtimeloading.swf file includes the automation.swc, automation_agent.swc, automation_dmv.swc, automation_flashflexkit.swc, and qtp.swc libraries.

2 Deploy the runtimeloading.swf, RunTimeLoading.html, and your application's SWF file to a web server. Request the RunTimeLoading.html file and pass the name of your SWF file as the value to the `automationswfurl` query string parameter. For example:

```
http://localhost/RunTimeLoading.html?automationswfurl=MyApp.swf
```

If you want to recompile the runtimeloading.swf file without the batch file, be sure to include automated testing support by adding the appropriate automation SWC files with the `include-libraries` compiler option.

The batch file for compiling the runtimeloading.swf file is Windows only. To compile the SWF file on Mac OS or Linux, you must write your own batch file.

Testing applications that load external libraries

Applications that load other SWF file libraries require a special setting for automated testing to function properly. A library that is loaded at run time (including run-time shared libraries (RSLs)) must be loaded into the Application-Domain of the loading application. If the SWF file used in the application is loaded in a different application domain, automated testing record and playback will not function properly.

The following example shows a library that is loaded into the same ApplicationDomain:

```
import flash.display.*;
import flash.net.URLRequest;
import flash.system.ApplicationDomain;
import flash.system.LoaderContext;

var ldr:Loader = new Loader();

var urlReq:URLRequest = new URLRequest("RuntimeClasses.swf");
var context:LoaderContext = new LoaderContext();
context.applicationDomain = ApplicationDomain.currentDomain;
loader.load(request, context);
```

Creating testable applications

As a Flex developer, there are some techniques that you can employ to make Flex applications as "test friendly" as possible. One of the most important tasks that you can perform is to make sure that objects are identifiable in the testing tool's scripts. This means that you should set the value of the `id` property for all controls that are tested, and ensure that you use a meaningful string for that `id` property. If you can use unique IDs for each control, the testing scripts are more readable.

Providing meaningful identification of objects

When working with testing tools such as QTP, a QC professional only sees the visual representation of objects in your application. A QC professional generally does not have access to the underlying code. When a QC professional records a script, it's very helpful to see IDs that help the tester identify the object clearly. You should take some time to understand how testing tools interpret Flex applications and determine what names to use for the test objects in the test scripts.

In most cases, testing tools use a visual cue, such as the label of a Button control, to identify the control in the script. Sometimes, however, testing tools use the Flex `id` property of an MXML tag to identify an object in the test script; if there is no value for the `id` property, testing tools use other properties, such as the `childIndex` property.

You should give all testable MXML components an ID to ensure that the test script has a unique identifier to use when referring to that Flex control. You should also try to make these identifiers as human-readable as possible to make it easier for a QC professional to identify that object in the testing script. For example, set the `id` property of a Panel container inside a TabNavigator to *submit_panel* rather than *panel1* or *p1*.

In some cases, agents do not use the `id` property, but it is a good practice to include it to avoid naming collisions or confusion. For more information about how QTP identifies Flex objects, see *Testing Adobe Flex Applications with Mercury QuickTest Professional*.

You should set the value of the `automationName` property for all objects that are part of the application's test. The value of this property appears in the testing scripts. Providing a meaningful name makes it easier for QC professionals to identify that object. For more information about using the `automationName` property, see "Setting the automationName property" on page 371.

Avoiding duplication of objects

Automation agents rely on the fact that some properties of object instances should not be changed during run time. If you change the Flex component property that is used by the agent as the object name at run time unexpected results can occur.

For example, if you create a Button control without an `automationName` property, and you do not initially set the value of its `label` property, and then later set the value of the `label` property, the agent might get confused. This is because agents often use the value of the `label` property of Button controls to identify an object in its object repository if the `automationName` property is not set. If you later set the value of the `label` property, or change the value of an existing `label` while the QC professional is recording a test, an automation tool such as QTP will create a new object in its repository instead of using the exising reference.

As a result, you should try to understand what properties are used to identify objects in the agent, and try to avoid changing those properties at run time. You should set unique, human-readable `id` or `automationName` properties for all objects that are included in the recorded script.

Coding containers

Containers are different from other kinds of controls because they are used both to record user interactions (such as when a user moves to the next pane in an Accordion container) and to provide unique locations for controls in the testing scripts.

Adding and removing containers from the automation hierarchy

In general, the automated testing feature reduces the amount of detail about nested containers in its scripts. It removes containers that have no impact on the results of the test or on the identification of the controls from the script. This applies to containers that are used exclusively for layout, such as the HBox, VBox and Canvas containers, except when they are being used in multiple-view navigator containers such as the ViewStack, TabNavigator or Accordion containers. In these cases, they are added to the automation hierarchy to provide navigation.

Many composite components use containers, such as Canvas or VBox, to organize their children. These containers do not have any visible impact on the application. So, you usually exclude these containers from being tested because there is no user interaction and no visual need for their operations to be recordable. By excluding a container from being tested, it does not clutter the test scripts and make them harder to read.

To exclude a container from being recorded (but not exclude its children), set the container's `showInAutomationHierarchy` property to `false`. This property is defined by the UIComponent class, so all containers that subclass UIComponent have this property. Children of containers that are not visible in the hierarchy appear as children of the next highest visible parent.

The default value of the `showInAutomationHierarchy` property depends on the type of container. For containers such as Panel, Accordion, Application, DividedBox, and Form, the default value is `true`; for other containers, such as Canvas, HBox, VBox, and FormItem, the default value is `false`.

The following example forces the VBox containers to be included in the test script's hierarchy:

```
<?xml version="1.0"?>
<!-- at/NestedButton.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Panel title="ComboBox Control Example">
        <mx:HBox id="hb">
            <mx:VBox id="vb1" showInAutomationHierarchy="true">
                <mx:Canvas id="c1">
                    <mx:Button id="b1"
                        automationName="Nested Button 1"
                        label="Click Me"
                    />
                </mx:Canvas>
            </mx:VBox>
            <mx:VBox id="vb2" showInAutomationHierarchy="true">
                <mx:Canvas id="c2">
                    <mx:Button id="b2"
                        automationName="Nested Button 2"
                        label="Click Me 2"
                    />
                </mx:Canvas>
            </mx:VBox>
        </mx:HBox>
    </mx:Panel>
</mx:Application>
```

Working with multiview containers

You should avoid using the same label on multiple tabs in multiview containers, such as TabNavigator and Accordion containers. Although it is possible to use the same labels, this is generally not an acceptable UI design practice and can cause problems with control identification in your testing environment. QTP, for example, uses the `label` properties to identify those views to testers. When two labels are the same, QTP uses different strategies to uniquely identify the tabs, which can result in a confusing name list.

Also, dynamically adding children to multiview containers can cause delays that might confuse the testing tool. You should try to avoid this.

Understanding the automation framework

The automation interfaces and the flow of the automation framework change as you initialize, record, and play back an automatable event.

About the automation interfaces

The Flex class hierarchy includes the following interfaces in the mx.automation.* package that enable automation:

Interface	Description
IAutomationClass	Defines the interface for a component class descriptor.
IAutomationEnvironment	Provides information about the objects and properties of automatable components needed for communicating with agents.
IAutomationEventDescriptor	Defines the interface for an event descriptor.
IAutomationManager	Defines the interface expected from an AutomationManager by the automation module.
IAutomationMethodDescriptor	Defines the interface for a method descriptor.
IAutomationObject	Defines the interface for a delegate object implementing automation for a component.
IAutomationObjectHelper	Provides helper methods for the IAutomationObject interface.
IAutomationPropertyDescriptor	Describes a property of a test object as well as properties of an event object.

About the IAutomationObjectHelper

The IAutomationObjectHelper interface helps the components accomplish the following tasks:

- Replay mouse and keyboard events; the helper generates proper sequence of player level mouse and key events.
- Generate AutomationIDPart for a child: AutomationIDPart would be requested by the Automation for representing a component instance to agents.
- Find a child matching a AutomationIDPart: Automation would request the component to locate a child matching the AutomationIDPart supplied by an agent to it.
- Avoid synchronization issues: Agents invoke methods on Automation requesting operations on components in a sequence. Components may not be ready all the time to perform operations.

For example, an agent can invoke comboBox.Open, comboBox.select "Item1" operations in a sequence. Because it takes time for the drop-down list to open and initialize, it is not possible to run the select operation immediately. You can place a wait request during the open operation execution. The wait request should provide a function for automation, which can be invoked to check the ComboBox control's readiness before invoking the next operation.

Automated testing workflow with the QTP automation tool

Before you automate custom components, you might find it helpful to see the order of events during which Flex's automation framework initializes, records, and plays back events with QTP. You should keep in mind that the QTP adapter class' implementation is only one way to use the automation API for automated testing.

Automated testing initialization

1 The user launches the Flex application. Automation initialization code associates component delegate classes with component classes. Component delegate classes implement the IAutomationObject interface.

2 AutomationManager is a mixin. Its instance is created in the mixin `init()` method.

3 The SystemManager initializes the application. Component instances and their corresponding delegate instances are created. Delegate instances add event listeners for events of interest.

4 QTPAgent class is a mixin. In its `init()` method, it registers itself for the `FlexEvent.APPLICATION_COMPLETE` event which is dispatched from the SystemManager. On receiving the event, it creates a QTPAdapter object.

5 QTPAdapter sets up the ExternalInterface function map. QTPAdapter loads the QTP Plugin DLLs by creating the ActiveX object to communicate with QTP.

6 The QTPAdapter requests the XML environment information from the plugin and passes it to the AutomationManager.

7 The XML information is stored in a chain of AutomationClass, AutomationMethodDescriptor, and AutomationPropertyDescriptor objects.

Automated testing recording

1 The user clicks the Record button in QTP.

2 QTP calls the `QTPAdapter.beginRecording()` method. QTPAdapter adds a listener for `AutomationRecordEvent.RECORD` from the AutomationManager.

3 The QTPAdapter notifies AutomationManager about this by calling the `beginRecording()` method. The AutomationManager adds a listener for the `AutomationRecordEvent.RECORD` event from the SystemManager.

4 The user interacts with the application. In this example, suppose the user clicks a Button control.

5 The `ButtonDelegate.clickEventHandler()` method dispatches an AutomationRecordEvent event with the `click` event and Button instance as properties.

6 The AutomationManager `record` event handler determines which properties of the `click` event to store, based on the XML environment information. It converts the values into proper type or format. It dispatches the `record` event.

7 The QTPAdapter event handler receives the event. It calls the `AutomationManager.createID()` method to create the AutomationID object of the button. This object provides a structure for object identification.

The AutomationID structure is an array of AutomationIDParts. An AutomationIDPart is created by using IAutomationObject. (The `UIComponent.id`, `automationName`, `automationValue`, `childIndex`, and `label` properties of the Button control are read and stored in the object. The `label` property is used because the XML information specifies that this property can be used for identification for the Button.)

8 The QTPAdapter uses the `AutomationManager.getParent()` method to get the logical parent of the Button control. The AutomationIDPart objects of parent controls are collected at each level up to the application level.

9 All these AutomationIDParts are made part of an AutomationID object.

10 The QTPAdapter sends the information in a call to QTP.

11 At this point, QTP might call the `AutomationManager.getProperties()` method to get the property values of the Button control. The property type information and codec that should be used to modify the value format are gotten from the AutomationPropertyDescriptor.

12 User stops recording. This is propagated by a call to the `QTPAdapter.endRecording()` method.

Automated testing playback

1 The user clicks the Playback button in QTP.

2 The `QTPAdapter.findObject()` method is called to determine whether the object on which the event has to be played back can be found. The AutomationID object is built from the XML data received. The `AutomationManager.resolveIDToSingleObject()` method is invoked to see if QTP can find one unique object matching the AutomationID. The `AutomationManager.getChildren()` method is invoked from application level to find the child object. The `IAutomationObject.numAutomationChildren` property and the `IAutomationObject.getAutomationChildAt()` method are used to navigate the application.

3 The `AutomationManager.isSynchronized()` and `AutomationManager.isVisible()` methods ensure that the object is fully initialized and is visible so that it can receive the event.

4 QTP invokes the `QTPAdpater.run()` method to play back the event. The `AutomationManager.replayAutomatableEvent()` method is called to replay the event.

5 The AutomationMethodDescriptor for the `click` event on the Button is used to copy the property values (if any).

6 The `AutomationManager.replayAutomatableEvent()` method invokes the `IAutomationObject.replayAutomatableEvent()` method on the delegate class. The delegate uses the `IAutomationObjectHelper.replayMouseEvent()` method (or one of the other replay methods, such as `replayKeyboardEvent()`) to play back the event.

7 If there are check points recorded in QTP, the `AutomationManager.getProperties()` method is invoked to verify the values.

Instrumenting events

When you extend Flex components that are already instrumented, you do not have to change anything to ensure that those components' events can be recorded by a testing tool. For example, if you extend a Button class, the class still dispatches the automation events when the Button is clicked, unless you override the Button control's default event dispatching behavior.

Automation events (sometimes known in automation tools such as QTP as *operations*) are not the same as Flex events. Flex must dispatch an automation event as a separate action. Flex dispatches them at the same time as Flex events, and uses the same event classes, but you must decide whether to make a Flex event visible to the automation tool.

Not all events on a control are instrumented. You can instrument additional events by using the instructions in "Instrumenting existing events" on page 365.

If you change the instrumentation of a component, you must edit that component's entry in the TEAFlex.xml file. This is described in "Using the class definitions file" on page 369.

Instrumenting existing events

Events have different levels of relevance for the QC professional. For example, a QC professional is generally interested in recording and playing back a `click` event on a Button control. The QC professional is not generally interested in recording all the events that occur when a user clicks the Button, such as the `mouseOver`, `mouseDown`, `mouseUp`, and `mouseOut` events. For this reason, when a tester clicks on a Button control with the mouse, testing tools only record and play back the `click` event for the Button control and not the other lower-level events.

There are some circumstances where you would want to record events that are normally ignored by the testing tool. But the testing tool's object model only records events that represent the end-user's gesture (such as a click or a drag and drop). This makes a script more readable and it also makes the script robust enough so that it does not fail if you change the application slightly. So, you should carefully consider whether you add a new event to be tested or you can rely on events in the existing object model.

You can see a list of events that the QTP automation tool can record for each Flex component in the *QTP Object Type Information* document. The Button control, for example, supports the following operations:

- `ChangeFocus`
- `Click`
- `MouseMove`
- `SetFocus`
- `Type`

All of these events except for `MouseMove` are automatically recorded by QTP by default. The QC professional must explicitly add the `MouseMove` event to their QTP script for QTP to play back the event.

However, you can alter the behavior of your application so that this event is recorded by the testing tool. To add a new event to be tested, you override the `replayAutomatableEvent()` method of the IAutomationObject interface. Because UIComponent implements this interface, all subclasses of UIComponent (which include all visible Flex controls) can override this method. To override the `replayAutomatableEvent()` method, you create a custom class, and override the method in that class.

The `replayAutomatableEvent()` method has the following signature:

```
public function replayAutomatableEvent(event:Event):Boolean
```

The *`event`* argument is the Event object that is being dispatched. In general, you pass the Event object that triggered the event. Where possible, you pass the specific event, such as a MouseEvent, rather than the generic Event object.

The following example shows a custom Button control that overrides the `replayAutomatableEvent()` method. This method checks for the `mouseMove` event and calls the `replayMouseEvent()` method if it finds that event. Otherwise, it calls its superclass' `replayAutomatableEvent()` method.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- at/CustomButton.mxml -->
<mx:Button xmlns:mx="http://www.adobe.com/2006/mxml">
        <mx:Script>
        <![CDATA[
            import flash.events.Event;
            import flash.events.MouseEvent;
            import mx.automation.Automation;
            import mx.automation.IAutomationObjectHelper;

            override public function
                replayAutomatableEvent(event:Event):Boolean {

                trace('in replayAutomatableEvent()');

                var help:IAutomationObjectHelper =
                    Automation.automationObjectHelper;

                if (event is MouseEvent &&
                    event.type == MouseEvent.MOUSE_MOVE) {
                    return help.replayMouseEvent(this, MouseEvent(event));
                } else {
                    return super.replayAutomatableEvent(event);
                }
            }
        ]]>
    </mx:Script>
</mx:Button>
```

In the application, you call the AutomationManager's `recordAutomatableEvent()` method when the user moves the mouse over the button. The following application uses this custom class:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- agent/CustomButtonApp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns:ns1="*"
initialize="doInit()">
    <mx:Script>
    <![CDATA[
        import mx.automation.*;

        public function doInit():void {
            b1.addEventListener(MouseEvent.MOUSE_MOVE,
                dispatchLowLevelEvent);
        }
```

```
        public function dispatchLowLevelEvent(e:MouseEvent):void {
            var help:IAutomationManager = Automation.automationManager;
            help.recordAutomatableEvent(b1,e,false);
        }
    ]]>
    </mx:Script>

    <ns1:CustomButton id="b1"
        toolTip="Mouse moved over"
        label="CustomButton"
    />

</mx:Application>
```

If the event is not one that is currently recordable, you also must define the new event for the agent. For QTP, the class definitions are in the TEAFlex.xml file. Automation tools can use files like this to define the events, properties, and arguments for each class of test object. For more information, see "Instrumenting events" on page 365. For example, if you wanted to add support for the `mouseOver` event, you would add the following to the FlexButton's entry in the TEAFlex.xml file:

```
<Operation Name="MouseOver" PropertyType="Method" ExposureLevel="CommonUsed">
    <Implementation Class="flash.events::MouseEvent" Type="mouseOver"/>
        <Argument Name="keyModifier" IsMandatory="false" DefaultValue="0">
            <Type VariantType="Enumeration"
                ListOfValuesName="FlexKeyModifierValues"
                Codec="keyModifier"
            />
            <Description>Occurs when the user moves mouse over the
                component</Description>
        </Argument>
</Operation>
```

In the preceding example, however, the `mouseMove` event is already in the FlexButton control's entry in that file, so no editing is necessary. The difference now is that the QC professional does not have to explicitly add the event to their script. After you compile this application and deploy the new TEAFlex.xml file to the QTP testing environment, QTP records the `mouseMove` event for all of the CustomButton objects.

For more information on using a class definitions file, see "Using the class definitions file" on page 369.

Instrumenting custom components

The process of creating a custom component that supports automated testing is called instrumentation. Flex framework components are instrumented by attaching a delegate class to each component at run time. The *delegate class* defines the methods and properties required to perform instrumentation.

If you extend an existing component that is instrumented, such as a Button control, you inherit its parent's instrumentation, and are not required to do anything else to make that component testable. If you create a component that inherits from UIComponent, you must instrument that class in one of the following ways:

- Create a delegate class that implements the required interfaces.
- Add testing-related code to the component.

You usually instrument components by creating delegate classes. You can also instrument components by adding automation code inside the components, but this is not a recommended practice. It creates tighter coupling between automated testing code and component code, and it forces the automated testing code to be included in a production SWF file.

In both methods of instrumenting a component, you must specify any new events to the agent. For QTP, you must add your new component's information to a class definitions XML file so that QTP recognizes that component. For more information about this file, see "Using the class definitions file" on page 369.

Consider the following additional factors when you instrument custom components:

- Composition. When instrumenting components, you must consider whether the component is a simple component or a composite component. Composite components are components made up of several other components. For example, a TitleWindow that contains form elements is a composite component.
- Container hierarchy. You should understand how containers are viewed in the automation hierarchy so that the QC professional can easily test the components. Also, you should be aware that you can manipulate the hierarchy to better suit your application by setting some automation-related properties.
- Automation names. Custom components sometimes have ambiguous or unclear default automation names. The ambiguous name makes it more difficult in automation tools to determine what component a script is referring to. Component authors can manually set the value of the `automationName` property for all components except item renderers. For item renderers, use the `automationValue`.

Creating a delegate class

To instrument custom components with a delegate, you must do the following:

- Create a delegate class that implements the required interfaces. In most cases, you extend the UIComponentAutomationImpl class. You can instrument any component that implements IUIComponent.
- Register the delegate class with the AutomationManager.
- Define the component in a class definitions XML file.

The delegate class is a separate class that is not embedded in the component code. This helps to reduce the component class size and also keeps automated testing code out of the final production SWF file. All Flex controls have their own delegate classes. These classes are in the mx.automation.delegates.* package. The class names follow a pattern of *Classname*AutomationImpl. For example, the delegate class for a Button control is mx.automation.delegates.controls.ButtonAutomationImpl.

Instrument with a delegate class

1. Create a delegate class.
2. Mark the delegate class as a mixin by using the `[Mixin]` metadata keyword.
3. Register the delegate with the AutomationManager by calling the `AutomationManager.registerDelegateClass()` method in the `init()` method. The following code is a simple example:

```
[Mixin]
public class MyCompDelegate {
public static init(root:DisplayObject):void {
// Pass the component and delegate class information.
AutomationManager.registerDelegateClass(MyComp, MyCompDelegate);
}
}
```

   You pass the custom class and the delegate class to the `registerDelegateClass()` method.

4. Add the following code to your delegate class:

   a Override the getter for the `automationName` property and define its value. This is the name of the object as it usually appears in automation tools such as QTP. If you are defining an item renderer, use the `automationValue` property instead.

b Override the getter for the `automationValue` property and define its value. This is the value of the object in automation tools such as QTP.

c In the constructor, add event listeners for events that the automation tool records.

d Override the `replayAutomatableEvent()` method. The AutomationManager calls this method for replaying events. In this method, return whether the replay was successful. You can use methods of the helper classes to replay common events.

For examples of delegates, see the source code for the Flex controls in the mx.automation.delegates.* packages.

5 Link the delegate class with the application SWF file in one of these ways:

- Add the following `includes` compiler option and link in the delegate class:

```
mxmlc -includes MyCompDelegate -- FlexApp.mxml
```

- Build a SWC file for the delegate class by using the compc component compiler:

```
compc -source-path+=. -include-classes MyCompDelegate -output MyComp.swc
```

Then include this SWC file with your Flex application by using the following `include-libraries` compiler option:

```
mxmlc -include-libraries MyComp.SWC -- FlexApp.mxml
```

This approach is useful if you have many components and delegate classes and want to include them as a single file.

6 After you compile your Flex application with the new delegate class, you must define the new interaction for the agent and the automation tool. For QTP, you must add the new component to QTP's custom class definition XML file. For more information, see "Using the class definitions file" on page 369.

For an example that shows how to instrument a custom component, see "Example: Instrumenting the RandomWalk custom component for QTP" on page 375.

Using the class definitions file

The class definitions file contains information about all instrumented Flex components. This file provides information about the components to the automation agent, including what events can be recorded and played back, the name of the component, and the properties that can be tested.

An example of a class definitions file is the TEAFlex.xml file, which is specific to the QTP automation tool. This file is included in the Flex Automation Package. Another example is the FlexEnv.xml file included in the AutoQuick example. These files are very similar, but represent only one way that class definitions can be represented to the agent and automation tool.

The TEAFlex.xml file is located in the "*QTP_plugin_install*\Flex 3 Plug-in for Mercury QuickTest Pro" directory. QTP recognizes any file in that directory that matches the pattern TEAFlex*.xml, where * can be any string. This directory also contains a TEAFlexCustom.xml file that you can use as a starting point for adding custom component definitions.

The TEAFlex.xml and FlexEnv.xml class definitions files describe instrumented components with the following basic structure:

```
<TypeInformation>
    <ClassInfo>
        <Description/>
        <Implementation/>
        <TypeInfo>
            <Operation/>
            ...
        </TypeInfo>
```

```
        <Properties>
            <Property/>
            ...
        </Properties>
    </ClassInfo>
</TypeInformation>
```

The top level tag is `<TypeInformation>`. You define a new class that uses the `<ClassInfo>` tag, which is a child tag of the `<TypeInformation>` tag. The `<ClassInfo>` tag has child tags that further define the instrumented classes. The following table describes these tags:

Tag	Description
ClassInfo	Defines the class that is instrumented, for example, FlexButton. This is the name that the automation tools use for the Button control. Attributes of this tag include `Name`, `GenericTypeID`, `Extends`, and `SupportsTabularData`.
Description	Defines the text that appears in the automation tool to define the component. This is not implemented in the AutoQuick example, but is implemented for QTP.
Implementation	Defines the class name, as it is known by the Flex compiler, for example, Button or MyComponent.
TypeInfo	Defines events for this class. Each event is defined in an `<Operation>` child tag, which has two child tags: • The `<Implementation>` child tag associates the operation with the actual event. • Each operation can also define properties of the event object by using an `<Argument>` child tag.
Properties	Defines properties of the class. Each property is defined in a `<Property>` child tag. Inside this tag, you define the property's type, name, and description. For each `Property`, if the `ForDescription` attribute is `true`, the property is used to uniquely identify a component instance in the automation tool; for example, the `label` property of a Button control. QTP lists this property as part of the object in QTP object repository. If the `ForVerfication` attribute is `true`, the property is visible in the properties dialog box in QTP. If the `ForDefaultVerification` tag is `true`, the property appears selected by default in the dialog box in QTP. This results in verification of the property value in the checkpoint.

The following example adds a new component, MyComponent, to the class definition file. This component has one instrumented event, click:

```
<TypeInformation xsi:noNamespaceSchemaLocation="ClassesDefintions.xsd" Priority="0"
PackageName="TEA" Load="true" id="Flex" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
    <ClassInfo Name="MyComponent" GenericTypeID="mycomponent"
        Extends="FlexObject" SupportsTabularData="false">
        <Description>FlexMyComponent</Description>
        <Implementation Class="MyComponent"/>
        <TypeInfo>
            <Operation Name="Select" PropertyType="Method"
                ExposureLevel="CommonUsed">
                <Implementation Class="myComponentClasses::MyComponentEvent"
                    Type="click"/>
            </Operation>
        </TypeInfo>
        <Properties>
            <Property Name="automationClassName" ForDescription="true">
                <Type VariantType="String"/>
                <Description>This is MyComponent.</Description>
            </Property>
            <Property Name="automationName" ForDescription="true">
                <Type VariantType="String"/>
                <Description>The name used by tools to id an object.</Description>
```

```
            </Property>
            <Property Name="className" ForDescription="true">
                <Type VariantType="String"/>
                <Description>To be written.</Description>
            </Property>
            <Property Name="id" ForDescription="true" ForVerification="true">
                <Type VariantType="String"/>
                <Description>Developer-assigned ID.</Description>
            </Property>
            <Property Name="index" ForDescription="true">
                <Type VariantType="String"/>
                <Description>The index relative to its parent.</Description>
            </Property>
        </Properties>
    </ClassInfo>
    ...
</TypeInformation>
```

You can edit the class definitions file to add a new recordable event to an existing component. To do this, you insert a new `<Operation>` in the control's `<TypeInfo>` block. This includes the implementation class of the event, and any arguments that the event might take.

The following example adds a new event, `MouseOver`, with several arguments to the Button control:

```
<TypeInfo>
    <Operation ExposureLevel="CommonUsed" Name="MouseOver" PropertyType="Method">
        <Implementation Class="flash.events::MouseEvent" Type="mouseOver"/>
        <Argument Name="inputType" IsMandatory="false"
            DefaultValue="mouse">
            <Type VariantType="String"/>
        </Argument>
        <Argument Name="shiftKey" IsMandatory="false" DefaultValue="false">
            <Type VariantType="Boolean"/>
        </Argument>
        <Argument Name="ctrlKey" IsMandatory="false" DefaultValue="false">
            <Type VariantType="Boolean"/>
        </Argument>
        <Argument Name="altKey" IsMandatory="false" DefaultValue="false">
            <Type VariantType="Boolean"/>
        </Argument>
    </Operation>
</TypeInfo>
```

When you finish editing the class definitions file, you must distribute the new file to the automation tool users. For QTP, users must copy this file manually to the "*QTP_plugin_install*\Flex 3 Plug-in for Mercury QuickTest Pro" directory. When you replace the class definitions file in the QTP environment, you must restart QTP.

Setting the automationName property

The `automationName` property defines the name of a component as it appears in testing scripts. The default value of this property varies depending on the type of component. For example, a Button control's `automationName` is the label of the Button control. Sometimes, the `automationName` is the same as the control's `id` property, but this is not always the case.

For some components, Flex sets the value of the `automationName` property to a recognizable attribute of that component. This helps QC professionals recognize that component in their scripts. Because they do not usually have access to the underlying source code of the application, having a control's visible property define that control can be useful. For example, a Button labeled "Process Form Now" appears in the testing scripts as `FlexButton("Process Form Now")`.

If you implement a new component, or derive from an existing component, you might want to override the default value of the `automationName` property. For example, UIComponent sets the value of the `automationName` to the component's `id` property by default, but some components use their own methods of setting its value.

For example, in the Flex Store sample application, containers are used to create the product thumbnails. A container's default `automationName` (it is the same as the container's `id` property) would not be very useful because it is programmatically generated. So in Flex Store, the custom component that generates a product thumbnail explicitly sets the `automationName` to the product name to make testing the application easier.

The following example from the CatalogPanel.mxml custom component sets the value of the `automationName` property to the name of the item as it appears in the catalog. This is much more recognizable than the default automation name.

```
thumbs[i].automationName = catalog[i].name;
```

The following example sets the `automationName` property of the ComboBox control to "Credit Card List"; rather than using the `id` property, the testing tool typically uses "Credit Card List" to identify the ComboBox in its scripts:

```
<?xml version="1.0"?>
<!-- at/SimpleComboBox.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            [Bindable]
            public var cards: Array = [
                {label:"Visa", data:1},
                {label:"MasterCard", data:2},
                {label:"American Express", data:3}
            ];

            [Bindable]
            public var selectedItem:Object;
        ]]>
    </mx:Script>
    <mx:Panel title="ComboBox Control Example">
        <mx:ComboBox id="cb1" dataProvider="{cards}"
            width="150"
            close="selectedItem=ComboBox(event.target).selectedItem"
            automationName="Credit Card List"
        />

        <mx:VBox width="250">
            <mx:Text
                width="200"
                color="blue"
                text="Select a type of credit card."
            />
            <mx:Label text="You selected: {selectedItem.label}"/>
            <mx:Label text="Data: {selectedItem.data}"/>
        </mx:VBox>
    </mx:Panel>
</mx:Application>
```

If you do not set the value of the `automationName` property, the name of an object in a testing tool is sometimes a property that can change while the application runs. If you set the value of the `automationName` property, testing scripts use that value rather than the default value. For example, by default, QTP uses a Button control's `label` property as the name of the Button in the script. If the label changes, the script can break. You can prevent this from happening by explicitly setting the value of the `automationName` property.

Buttons that have no label, but have an icon, are recorded by their index number. In this case, you should ensure that you set the `automationName` property to something meaningful so that the QC professional can recognize the Button in the script. This might not be necessary if you set the `toolTip` property of the Button because QTP uses that value if there is no label.

After the value of the `automationName` property is set, you should never change the value during the component's life cycle.

For item renderers, use the `automationValue` property rather than the `automationName` property. You do this by overriding the `createAutomationIDPart()` method and returning a new value that you assign to the `automationName` property, as the following example shows:

```
<mx:List xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            import mx.automation.IAutomationObject;
            override public function
                createAutomationIDPart(item:IAutomationObject):Object {
                    var id:Object = super.createAutomationIDPart(item);
                    id["automationName"] = id["automationIndex"];
                    return id;
            }
        ]]>
    </mx:Script>
</mx:List>
```

This technique works for any container or list-like control to add index values to their children. There is no method for a child to specify an index for itself.

Instrumenting composite components

Composite components are custom components made up of two or more components. A common composite component is a form that contains several text fields, labels, and buttons. Composite components can be MXML files or ActionScript classes.

By default, you can record operations on all instrumented child controls of a container. If you have a Button control inside a custom TitleWindow container, the QA professional can record actions on that Button control just like on any Button control. You can, however, create a composite component in which some of the child controls are instrumented and some are not. To prevent the operations of a child component from being recorded, you override the following methods:

- `numAutomationChildren` getter
- `getAutomationChildAt()`

The `numAutomationChildren` property is a read-only property that stores the number of automatable children that a container has. This property is available on all containers that delegate implementation classes. To exclude some children from being automated, you return a number that is less than the total number of children.

The `getAutomatedChildAt()` method returns the child at the specified index. When you override this method, you return null for the unwanted child at the specified index, but return the other children as you normally would.

The following custom composite component is written in ActionScript. It consists of a VBox container with three buttons (OK, Cancel, and Help). You cannot record the operations of the Help button. You can record the operations of the other Button controls, OK and Cancel. The following example sets the values of the OK and Cancel buttons' `automationName` properties. This makes those button controls easier to recognize in the automated testing tool's scripts.

```
// MyVbox.as
package { // Empty package
    import mx.core.UIComponent;
    import mx.containers.VBox;
    import mx.controls.Button;
    import mx.automation.IAutomationObject;
    import mx.automation.delegates.containers.BoxAutomationImpl;

    public class MyVBox extends VBox {
        public var btnOk : Button;
        public var btnHelp : Button;
        public var btnCancel : Button;

        public function MyVBox():void { // Constructor
        }

        override protected function createChildren():void {
            super.createChildren();

            btnOk = new Button();
            btnOk.label = "OK";
            btnOk.automationName = "OK_custom_form";
            addChild(btnOk);

            btnCancel = new Button();
            btnCancel.label = "Cancel";
            btnCancel.automationName = "Cancel_custom_form";
            addChild(btnCancel);

            btnHelp = new Button();
            btnHelp.label = "Help";
            btnHelp.showInAutomationHierarchy = false;
            addChild(btnHelp);
        }

        override public function get numAutomationChildren():int {
            return 2; //instead of 3
        }

        override public function
            getAutomationChildAt(index:int):IAutomationObject {
            switch(index) {
                case 0:
                    return btnOk;
                case 1:
                    return btnCancel;
            }
            return null;
        }
    } // Class
} // Package
```

To make this solution more portable, you could create a custom Button control and add a property that determines whether a Button should be testable. You could then set the value of this property based on the Button instance (for example, `btnHelp.useInAutomation = false`), and check against it in the overridden `getAutomationChildAt()` method, before returning null or the button instance.

Example: Instrumenting the RandomWalk custom component for QTP

The RandomWalk component is an example of a complex custom component. It has custom events and custom item renderers. Showing how it is instrumented can be helpful in instrumenting your custom components. For example, you can instrument the RandomWalk custom component so that interaction with it can be recorded and played back by using the QTP automation tool.

You can find the source code used in this section at the following location:

http://www.adobe.com/go/flex_automation_randomwalk_apps

Instrumenting the RandomWalk custom component

The first task when instrumenting a custom component is to create a delegate and add the new component to the class definitions XML file.

1 Create a RandomWalkDelegate class that extends UIComponentAutomationImpl, similar to RandomWalk extending the UIComponent class. The UIComponentAutomationImpl class implements the IAutomationObject interface.

2 Mark the delegate class as a mixin with the `[Mixin]` metadata tag; for example:

```
package {

    ...

[Mixin]
public class RandomWalkDelegate extends UIComponentAutomationImpl {
}
}
```

This results in a call to the static `init()` method in the class when the SWF file loads.

3 Add a public static `init()` method to the delegate class, and add the following code to it:

```
public static init(root:DisplayObject):void {
    Automation.registerDelegateClass(RandomWalk, RandomWalkDelegate);
}
```

4 Add a constructor that takes a RandomWalk object as parameter. Add a call to the super constructor and pass the object as the argument:

```
private var walker:RandomWalk
public function RandomWalkDelegate(randomWalk:RandomWalk) {
    super(randomWalk);
    walker = randomWalk;
}
```

5 Update the TEAFlexCustom.xml file so that QTP recognizes the RandomWalk component. To do this, add the text between the `<TypeInformation>` root tags. The TEAFlexCustom.xml file is located in the "*QTP_plugin_install*\Flex 3 Plug-in for Mercury QuickTest Pro" directory. For more information about the TEAFlexCustom.xml file, see "Using the class definitions file" on page 369.

```
<TypeInformation xsi:noNamespaceSchemaLocation="ClassesDefintions.xsd" Priority="0"
PackageName="TEA" Load="true" id="Flex" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
    ...
    <ClassInfo Name="FlexRandomWalk" GenericTypeID="randomwalk"
        Extends="FlexObject" SupportsTabularData="false">
        <Description>FlexRandomWalk</Description>
```

```
            <Implementation Class="RandomWalk"/>
            <TypeInfo>
            </TypeInfo>
            <Properties>
                <Property Name="automationClassName" ForDescription="true">
                    <Type VariantType="String"/>
                    <Description>To be written.</Description>
                </Property>
                <Property Name="automationName" ForDescription="true">
                    <Type VariantType="String"/>
                    <Description>The name used by the automation system to
                    identify an object.</Description>
                </Property>
                <Property Name="className" ForDescription="true">
                    <Type VariantType="String"/>
                    <Description>To be written.</Description>
                </Property>
                <Property Name="id" ForDescription="true" ForVerification="true">
                    <Type VariantType="String"/>
                    <Description>Developer-assigned ID.</Description>
                </Property>
                <Property Name="automationIndex" ForDescription="true">
                    <Type VariantType="String"/>
                    <Description>The object's index relative to its parent.
                    </Description>
                </Property>
            </Properties>
        </ClassInfo>
    </TypeInformation>
```

This defines the name of the FlexRandomWalk component and its implementing class. It specifies the `automationClassName`, `automationName`, `className`, `id` and `automationIndex` properties as available for identifying an instance of the RandomWalk class in the Flex application. This is possible because the RandomWalk class is derived from the UIComponent class, and these properties are defined on that parent class.

If your component has a property that you can use to differentiate between component instances, you can also add that property. For example, the `label` property of a Button control, though not unique when the whole application is considered, can be assumed to be unique within a container; therefore, you can use it as an identification property.

Instrumenting RandomWalk events

The next step in instrumenting a custom component is to identify the important events that must be recorded by QTP. The RandomWalk component dispatches a `RandomWalkEvent.ITEM_CLICK` event. Because this event indicates user navigation, it is important and must be recorded.

Instrument the ITEM_CLICK event

1 Add an event listener for the event in the RandomWalkDelegate constructor:

```
randomWalk.addEventListener(RandomWalkEvent.ITEM_CLICK, itemClickHandler)
```

2 Identify the event in the TEAFlexCustom.xml file so that QTP recognizes the event. Add the following text in the `<TypeInfo>` tag in the TEAFlexCustom.xml file:

```
<Operation Name="Select" PropertyType="Method"ExposureLevel="CommonUsed">
    <Implementation Class="randomWalkClasses::RandomWalkEvent"
        Type="itemClick"/>
</Operation>
```

Adding this block to the TEAFlexCustom.xml file names the event as Select and indicates that it is tied to the RandomWalkEvent class, which is available in the randomWalkClasses namespace. It also defines the event of type `itemClick`.

When using the RandomWalk component, users click on items to expand them. The component records the item label so that QC professionals can easily recognize their action in the QTP script.

The RandomWalkEvent class has only a single property, `item`, that stores the XML node information. In the RandomWalk component's implementation, the RandomWalkRenderer item renderer is used to display the data on the screen. This class is derived from the Label control, which is already instrumented. The Label control returns the label text as its `automationName`, which is what you want to record.

Record the label text

1 Add a new property, `itemRenderer`, to the RandomWalkEvent class.

2 Add the code to initialize this property for the event before dispatching the event; for example:

```
var rEvent:RandomWalkEvent = new RandomWalkEvent(RandomWalkEvent.ITEM_CLICK,node);
rEvent.itemRenderer = child as Label;
dispatchEvent(rEvent);
```

3 Add the new `itemRenderer` property as an argument to the Select operation in the TEAFlexCustom.xml file:

```
<Operation Name="Select" PropertyType="Method" ExposureLevel="CommonUsed">
    <Implementation Class="randomWalkClasses::RandomWalkEvent"
        Type="itemClick"/>
    <Argument Name="itemRenderer" IsMandatory="true" >
        <Type VariantType="String" Codec="automationObject"/>
        <Description>User-clicked item.</Description>
    </Argument>
</Operation>
```

This code block specifies the type of argument as String and the codec as automationObject. The `Codec` property is an optional attribute of the `<Type>` tag in the `<Argument>` and `<Property>` tags. It defines a class that converts ActionScript types to agent-specific types.

4 In the event handler, call the `recordAutomatableEvent()` method with `event` as the parameter, as the following example shows:

```
private function itemClickHandler(event:RandomWalkEvent):void {
    recordAutomatableEvent(event);
}
```

In some cases, the component does not dispatch any events or the event that was dispatched does not have the information that is required during the recording or playing back of the test script. In these circumstances, you must create an event class with the required properties. In the component, you create an instance of the event and pass it as a parameter to the `recordAutomatableEvent()` method.

For example, the `ListItemSelect` event has been added in automation code and is used by the List automation delegate to record and play back select operations for list items.

To locate the item renderer object, the automation classes require some help. Copy the standard implementation for the following methods:

```
override public function createAutomationIDPart(child:IAutomationObject):Object {
    var help:IAutomationObjectHelper = Automation.automationObjectHelper;
    return help.helpCreateIDPart(this, child);
}

override public function resolveAutomationIDPart(part:Object):Array {
    var help:IAutomationObjectHelper = Automation.automationObjectHelper;
    return help.helpResolveIDPart(this, part as AutomationIDPart);
```

```
}
```

Preparing the RandomWalk component for playback

Flex components display the data in a data provider after processing and formatting the data. Playback code requires a way to trace back the visual data to the data provider and the component displaying that data. For example, from the visual label of an item in the List, playback code should be able to find the item renderer that shows the element so that the item's click can be played back.

When playing back a RandomWalkEvent event, you must identify the item renderer with the `automationName` given. Because the RandomWalk component has many item renderers that are children that are derived from UIComponent subclasses, you must identify them by using the `automationName` property.

The IAutomationObject interface already has APIs to support this. The UIComponentAutomationImpl interface provides default implementations for some methods, but you must override some methods to return information that is specific to the RandomWalk component.

Prepare the RandomWalk component for playback

1 Instruct QTP as to how many renderers are being used by the instance of the RandomWalk component. RandomWalk uses an array of arrays for all the renderers. Add the following code in RandomWalkDelegate class to find the total number of instances:

```
override public function get numAutomationChildren():int {
    var numChildren:int = 0;
    var renderers:Array = walker.getItemRenderers();
    for (var i:int = 0;i< renderers.length;i++) {
        numChildren += renderers[i].length;
    }
    return numChildren;
}
```

2 Access the `itemRenderers` property in the delegate class; this property, however, is private. As a result, you must add the following accessor method to the RandomWalk component:

```
public function getItemRenderers():Array {
    return _renderers;
}
```

Alternatively, you can make the property public or change its namespace.

3 QTP requests each child renderer. Add the following code to determine the exact renderer and return it:

```
override public function getAutomationChildAt(index:int):IAutomationObject {
    var numChildren:int = 0;
    var renderers:Array = walker.getItemRenderers();
    for(var i:int = 0; i < renderers.length; i++) {
        if(index >= numChildren) {
            if(i+1 < renderers.length && (numChildren + renderers[i].length)
                <= index) {
                numChildren += renderers[i].length;
                continue;
            }
            var subIndex:int = index - numChildren;
            var instances:Array = renderers[i];
            return (instances[subIndex] as IAutomationObject);
        }
    }
    return null;
}
```

Linking the delegate to an application

There are two options to link the delegate class with the application SWF file. You can use the `includes` compiler option and link to the delegate as follows:

```
mxmlc -includes RandomWalkDelegate FlexApp.mxml
```

You can also build a SWC file for the delegate class. You then include the SWC file with the Flex application by using the `include-libraries` compiler option, as the following code shows:

```
mxmlc -include-libraries RandomWalkAT.SWC -- FlexApp.mxml
```

This approach is useful if you have many components and many delegate classes.

Adjusting event recording

You can compile and record any application that uses a RandomWalk component. While recording, you might notice that the `FlexLabel().Click` operation is recorded in addition to the `Select` operation. Generally, you do not want to record both operations for each user interaction.

In the following example, you stop the `AutomationRecordEvent.RECORD` event from being recorded. Because it is a bubbling event, you can listen to the event from the children, and prevent the event from being recorded.

Prevent an event from being recorded

1 Add an event handler in the constructor of the delegate, as the following example shows:

```
obj.addEventListener(AutomationRecordEvent.RECORD, labelRecordHandler);
```

2 Prevent the recording of this event by calling the `preventDefault()` method or by stopping the propagation of the event:

```
public function labelRecordHandler(event:AutomationRecordEvent):void {
    // if the event is not from the owning component reject it.
    if (event.replayableEvent.target != uiComponent)
        //event.preventDefault(); can also be used.
        event.stopImmediatePropagation();
}
```

The RandomWalkDelegate class must handle the playback of the RandomWalkEvent event only. Any other event must be handled by the super class implementation.

3 Override the `replayAutomatableEvent()` method and handle the RandomWalkEvent event:

```
override public function replayAutomatableEvent(event:Event):Boolean {
    if (event is RandomWalkEvent) {
    }
    return super.replayAutomatableEvent(event);
}
```

4 (Optional) To replay the RandomWalkEvent, you must replay a click on the item renderer. You do this by calling the `replayClick ()` method. To use the `replayClick()` method to play back a click on the item renderer, use the following code:

```
override public function replayAutomatableEvent(event:Event):Boolean {
    var help:IAutomationObjectHelper = Automation.automationObjectHelper;
    if (event is RandomWalkEvent) {
        var rEvent:RandomWalkEvent = event as RandomWalkEvent
        help.replayClick(rEvent.itemRenderer);
        return true;
    } else
        return super.replayAutomatableEvent(event);
    }
```

For playing back an event, the Automation.automationObjectHelper class providers some helper methods, the following table describes:

Method	Description
`replayClick()`	Dispatches the `mouseDown`, `mouseClick`, and `mouseUp` events on a UIComponent object.
`replayMouseEvent()`	Dispatches the specified mouse event on a UIComponent.
`replayKeyDownKeyUp()`	Dispatches a keyboard event with keyCode and key modifiers specified.
`replayKeyboardEvent()`	Dispatches the specified keyboard event on a UIComponent.

For more information, see the documentation for the IAutomationObjectHelper class in the *Adobe Flex Language Reference*.

5 Record and play back your application.

6 To ensure that the view is updated, add the following code after the call to the `replayClick()` method:

```
override public function replayAutomatableEvent(event:Event):Boolean {
    var help:IAutomationObjectHelper = Automation.automationObjectHelper;
    if (event is RandomWalkEvent) {
        var rEvent:RandomWalkEvent = event as RandomWalkEvent
        help.replayClick(rEvent.itemRenderer);
        (uiComponent as IInvalidating).validateNow();
        return true;
    } else
        return super.replayAutomatableEvent(event);
}
```

Adjustments like this might be required in the delegate to adjust the behavior of the component during playback because QTP does not wait for actions to be completed. The same can also be achieved by adding `Wait` statements in the QTP script.

You should now be able to compile, record, and play back any application with the RandomWalk component.

Adding checkpoints

To add checkpoints on public properties of the component, you add a small description of the property to the component description in the custom class definitions XML file (TEAFlexCustom.xml). You also add a getter for those public properties.

For the `openChildrenCount` property of the RandomWalk component to appear in checkpoints, add the following element as a child of the `<Properties>` tag:

```
<Property Name="openChildrenCount" ForVerification="true" ForDefaultVerification="true">
    <Type VariantType="Integer"/>
    <Description>Number of children open currently.</Description>
</Property>
```

When you use a checkpoint operation with a RandomWalk component, QTP displays the `openChildrenCount` property in the Checkpoint dialog box.

Also, add the following getter method to the RandomWalk class:

```
public function get openChildrenCount():int {
    return numAutomationChildren;
}
```

The `numAutomationChildren` property is inherited from the UIComponent class.

Chapter 9: Creating Custom Agents

Topics

About creating custom agents 381
About the automation APIs 382
Understanding the automation flow 385
Creating agents 387
Creating a recording agent 388
Creating a replaying agent 391

About creating custom agents

The automation API in Adobe® Flex® can be used for many tasks, including gathering metrics information, automated testing, and collaborative browsing (co-browsing).

To use the automation API, you should understand the following terminology:

- *agent* — An agent facilitates the communication between the Flex application and the automation tool. The way in which the communication happens depends on the automation tool, but it can include using the ExternalInterface API to access the browser's DOM or ActiveX plug-in. Agents are typically implemented in ActionScript and packaged as a SWC file.
- *automation tool* — An automation tool records, and sometimes plays back, interactions with Flex applications. An automation tool requires an agent to provide communication between it and the Flex application. Automation tools include Mercury QuickTest Professional™ (QTP) and Segue.

Full support for the Flex automation features is included in Adobe® Flex® Builder™ Professional. Adobe Flex Builder Standard allows only limited use of this feature.

To use the examples in this topic, you should download the customagent_src.zip and autoquick_src.zip files. The customagent_src.zip file includes a set of classes that define a custom automation agent that records user interaction with a Flex application. The interaction is then written out to a database. The autoquick_src.zip file records a user interaction and then plays it back. You can download these examples from the following location:

`http://www.adobe.com/go/flex_automation_agent_apps`

The sample files also include an XML file that the custom agents use to define their environment.

In addition to these samples, you can also download and run a testing-enabled version of the Adobe FlexStore application from http://www.adobe.com/go/flex_flexstore_automation.

The Automation Framework defines a single API that has two parts:

- Component API — Components must implement this API to support automation features. Developers can choose to put the code either in the main component itself or in a mixin class. Mixin classes implement this API for Flex components.
- Agent API — Agents use this API to communicate with the component API.

Component developers implement the component API for their component once and then the component is ready to converse with any agent. Agent developers implement the agent API for their specific feature or tool and it is able to work with any Flex application. For more information, see "About the automation APIs" on page 382.

Uses for agents

You can use agents to gather metrics information, to use automated testing, and to run applications at different locations at the same time.

Metrics

You might want to analyze how your online applications are being used. By gathering metrics information, you can answer the following questions:

- What product views are most popular?
- When do users abandon the checkout process?
- What is a typical path through my application?
- How many product views does a user look at during a session?

Automated testing

Maintaining the quality of a large software application is difficult. Verifying lots of functionality in any individual build can take a QA engineer many hours or even days, and much of the work between builds is repetitive. To alleviate this difficulty, automated testing tools have been created that can use applications and verify behavior without human intervention. Major application environments such as Java and .NET have testing tool support from vendors such as QTP and Segue.

By using the Flex automation API, you can:

- Record and replay events in a separate tool
- Manage communication between components and agents

Co-browsing

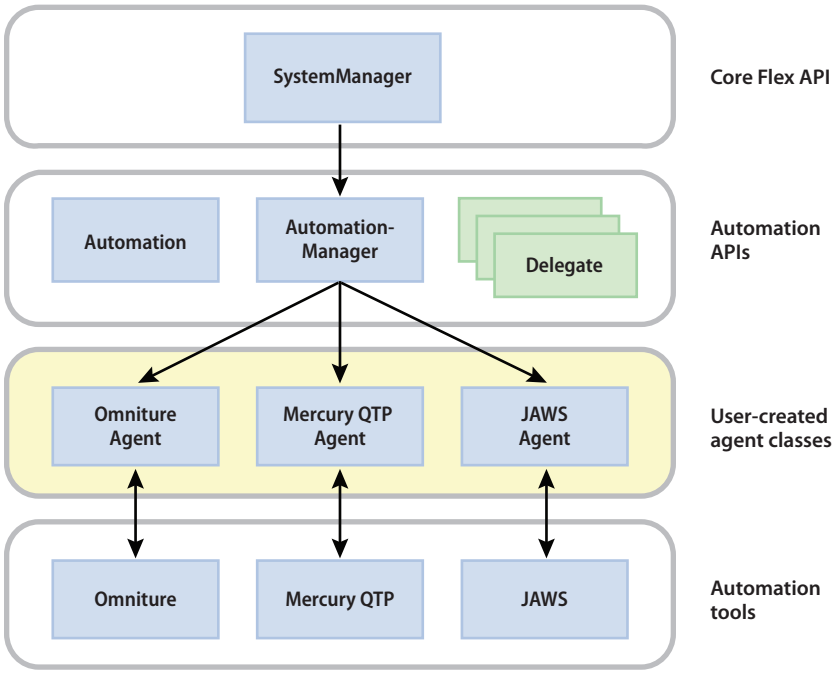
You might want to run the same application at different locations and view the application at the same time. By using the automation API, you can ensure that the applications are synchronized as users navigate through the them. User interaction at any location can be played at other locations and other users can see the action in real time.

About the automation APIs

There are four main parts that enable the automation framework in Flex:

- Core Flex API
- Automation APIs
- Agent class (also known as an adapter)
- Automation tools

The following illustration shows the relationship between these parts:



About the SystemManager class

The SystemManager class is one of the highest level classes in a Flex application. Every Flex application has a SystemManager class. It is the parent of all displayable objects in the application, such as the main mx.core.Application instance and all pop-up windows, ToolTip instances, cursors, and so on.

SystemManager calls the `init()` method on all mixins. The SystemManager is responsible for creating the AutomationManager class as well as the Agent and the delegate classes. SystemManager is also responsible for adding the Application object to Adobe® Flash® Player or Adobe® AIR™ stage (root).

About the AutomationManager class

The AutomationManager class is a Singleton that extends the EventDispatcher class. It implements the IAutomationManager, IAutomationObjectHelper, and IAutomationMouseSimulator interfaces.

AutomationManager is a mixin, so its `init()` method is called by the SystemManager class when the application is initialized. In the `init()` method, the AutomationManager class adds an event listener for the `Event.ADDED` event. This event is dispatched by Flash Player or AIR whenever a display object is added to the display list. When that happens, Flash Player or AIR calls the `childAddedHandler()` method:

```
root.addEventListener(Event.ADDED, childAddedHandler, false, 0, true);
```

In the `childAddedHandler()` method, the AutomationManager class:

- Creates a new instance of a delegate for each display object.
- Adds the display object to a delegate class map. This maps the display object to a delegate instance; for example:

  ```
  Automation.delegateClassMap[componentClass] = Automation.delegateClassMap[className];
  ```

  The delegate class map is a property of the Automation class.
- Ensures that all children of the new display object are added to the class map as well.

In the `recordAutomatableEvent()` method, the AutomationManager:

- Creates an `AutomationRecordEvent.RECORD` event.
- Dispatches the `RECORD` events. The agent listens for these events.

The `recordAutomatableEvent()` method is called by the delegates.

About the Automation class

During application initialization, the Automation class is created. This is a static class that maintains a map of its delegate class to its component class.

This class does the following for recording events:

- Provides access to the AutomationManager class
- Creates `delegateClassMap` as a static property

For example, there is a MyButton class that extends the Button class, but it does not have its own delegate class (MyButton might not add any new functionality to be recorded or played back). When AutomationManager encounters an instance of the MyButton class, it checks with the Automation class for a corresponding delegate class. When it fails to find one, it uses the `getSuperClassName()` method to get the super class of the MyButton class, which is the Button class.

AutomationManager then tries to find the delegate for this Button class. At that point, AutomationManager adds a new entry into the delegate-component class for the MyButton class, associating it with the ButtonDelegateAutomationImpl class, so that next time AutomationManager can find this mapping without searching the inheritance hierarchy.

About the delegate classes

The delegate classes provide automation hooks to the Flex framework and charting components. The delegate classes are in the mx.automation.delegates package, and they extend the UIComponentAutomationImpl class. The delegates are named *ControlName*AutomationImpl. For example, the Button control's delegate class is ButtonAutomationImpl.

The delegate classes register themselves with their associated Automation class by providing the component class and their own class as input. The AutomationManager class uses the Automation class-to-delegate class map to create a delegate instance that corresponds to a component instance in the `childAddedHandler()` method.

The delegate classes are mixins, so their `init()` method is called by the SystemManager class. The `init()` method of the delegate classes:

1 Calls the `registerDelegateClass()` method of the Automation class. This method maps the class to an automation component class; for example:

```
var className:String = getQualifiedClassName(compClass);
delegateClassMap[className] = delegateClass;
```

2 Adds event listeners for the mouse and keyboard events; for example:

```
obj.addEventListener(KeyboardEvent.KEY_UP, btnKeyUpHandler, false,
EventPriority.DEFAULT+1, true);
obj.addEventListener(MouseEvent.CLICK, clickHandler, false, EventPriority.DEFAULT+1,
true);
```

These event handlers call the AutomationManager class `recordAutomatableEvent()` method, which in turn dispatches the `AutomationRecordEvent.RECORD` events that the automation agent listens for.

All core framework and charting classes have delegate classes already created. You are not required to create any delegate classes unless you have custom components that dispatch events that you want to automate. In this case, you must create a custom delegate class for inclusion in your Flex application. For more information, see "Example: Instrumenting the RandomWalk custom component for QTP" on page 375.

About the agent

The agent facilitates communication between the Flex application and automation tools such as QTP and Segue.

When recording, the agent class is typically responsible for implementing a persistence mechanism in the automation process. It gets information about events, user sessions, and application properties and typically writes them out to a database, log file, LocalConnection, or some other persistent storage method.

When you create an agent, you compile it and its supporting classes into a SWC file. You then add that SWC file to your Flex application by using the `include-libraries` command-line compiler option.

The custom agent class must be a mixin, which means that its `init()` method is called by the SystemManager class. The `init()` method of the agent:

- Defines a handler for the `RECORD` events.
- Defines the environment. The environment indicates what components and their methods, properties, and events can be recorded with the automation API.

  The sample custom agent class uses an XML file that contains Flex component API information. The agent loads it with a call to the `URLRequest()` constructor, as the following example shows:

```
var myXMLURL:URLRequest = new URLRequest("AutomationGenericEnv.xml");
myLoader = new URLLoader(myXMLURL);
automationManager.automationEnvironment = new CustomEnvironment(new XML(source));
```

In this example, the source is an XML file that defines the Flex metadata (or environment information). This metadata includes the events and properties of the Flex components.

Note that representing events as XML is agent specific. The general-purpose automation API does not require it, but the XML file makes it easy to adjust the granularity of the events that are recorded.

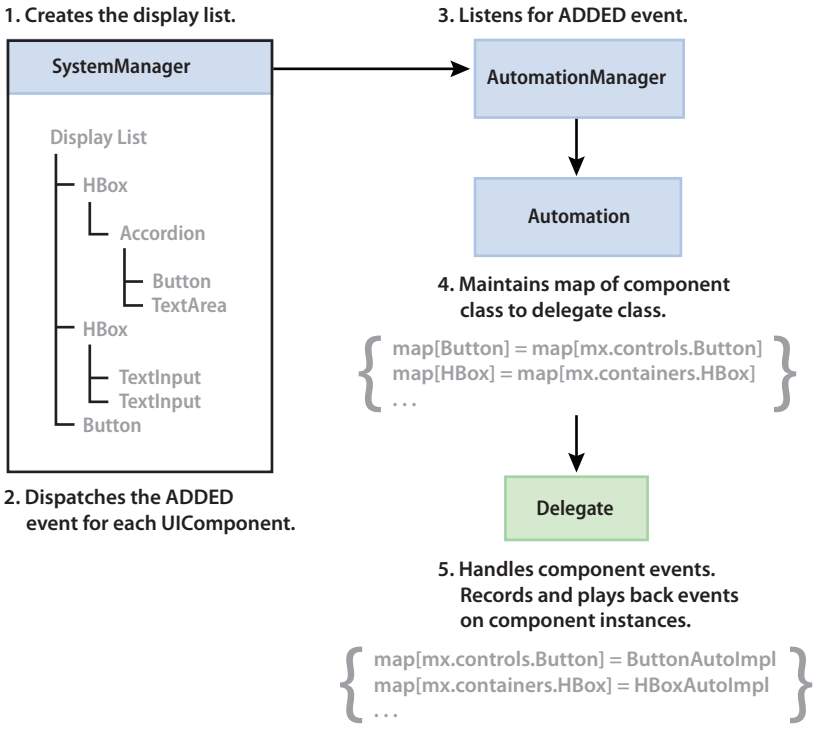
You are not required to create an instance of your adapter in the `init()` method. You can also create this instance in the `APPLICATION_COMPLETE` event handler if your agent requires that the application must be initialized before it is instantiated.

Most agents require a set of classes that handle the way in which the agent persists automation information and defines the environment. For more information, see "Creating a recording agent" on page 388.

Understanding the automation flow

When the application is initialized, the AutomationManager class is created. In its `init()` method, it adds a listener for `Event.ADDED` events.

The following image shows the order of events when the application is initialized and the AutomationManager class constructs the delegate map.



1 The SystemManager class creates the display list, a tree of visible objects that make up your application.

2 Each time a new component is added, either at the root of the display list or as a child of another member of the display list, SystemManager dispatches an `Event.ADDED` event.

3 AutomationManager listens for the `ADDED` event. In its `ADDED` event handler, it calls methods on the Automation class. It then instantiates the delegate for that class.

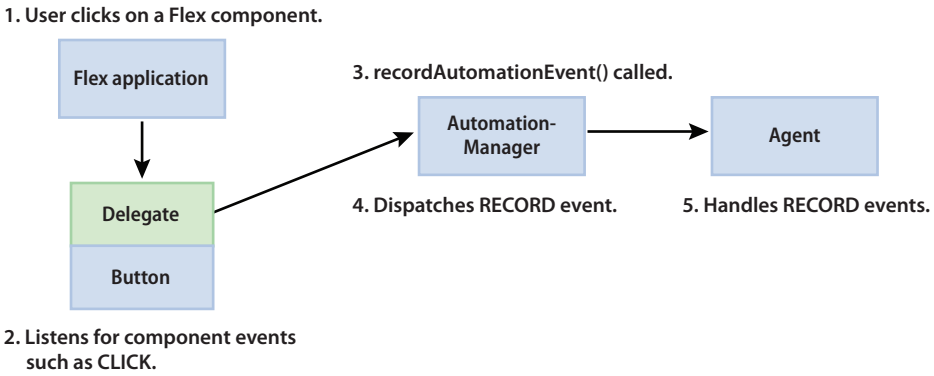
4 The Automation class maps each component in the display list to its full class name.

5 When it is created, the delegate class adds a reference to its instance in the delegate class map. The delegate class then handles events during record and play-back sequences.

The delegate is now considered *registered* with the component. It adds event listeners for the component's events and calls AutomationManager when the component triggers those events.

After the components in the display list are instantiated and mapped to instances of their delegate classes, AutomationManager is ready to listen for events and forward them to the agent for processing.

The following image shows the flow of operation when a user performs an action that is a recordable event. In this case, the user clicks a Button control in the application.



1 The user clicks the Button control in the Flex application. The SystemManager dispatches a MouseEvent.CLICK event.

2 The ButtonAutomationImpl class, the Button control's automation delegate, listens for `click` events. In the delegate's `click` event handler, the delegate calls the AutomationManager `recordAutomationEvent()` method. (It is likely that the button also defines a `click` event handler to respond to the user action, but that is not shown.)

3 The AutomationManager `recordAutomationEvent()` method dispatches an `AutomationRecordEvent.RECORD` event. In that event, the `replayableEvent` property points to the original `click` event.

4 The custom agent class listens for `RECORD` events. When it receives the `RECORD` event, it uses the `replayableEvent` property to access the properties of the original event.

5 The agent records the event properties in a database, logs the event properties, or gets information about the user before recording them.

Creating agents

You create an agent as a SWC file and link it into the Flex application by using the `include-libraries` compiler option. You can link multiple agents in any number of SWC files to the same Flex application. However, to use multiple agents at the same time, you must use the same environment configuration files for all agents.

The general process for creating a custom agent is:

- Mark the agent class as a mixin; this triggers a call to a static `init()` method from the SystemManager class on application start up.
- Get a reference to the AutomationManager class.
- Add event listeners for the `APPLICATION_COMPLETE` and `RECORD` events.
- Load the environment information (Flex metadata that describes the objects and operations of the Flex application). Environment information can be an XML file, as the samples use, or it can be in some other data format.
- Define a static `init()` method that creates an instance of the agent.
- Define a method that handles `RECORD` events. In that method, you can access:
  - Automation details such as the automation name
  - User information such as the FlexSession object (through a RemoteObject)

- The event that triggered the `RECORD` event
- The target object and its properties that triggered the `RECORD` event

The `RECORD` event handler in the agent gets an `AutomationRecordEvent` whose target is the object on which recording happened. The `automationManager.createID()` method converts the object to a string that can be recorded on the screen. Some tools may require the entire automation hierarchy that needs to be generated in this method.

You also use the custom agent class to enable and disable recording of automation events.

For more information on creating a custom agent, see "Creating a recording agent" on page 388.

Creating a recording agent

You can use the classes in the customagent_src.zip file to create a custom agent that records metrics data as a user interacts with an application. For information about creating an agent that records and plays back user interactions, see "Creating a replaying agent" on page 391.

The custom agent in the customagent_src.zip file is the CustomAdapter class. This class calls an HTTPService that writes event data out to a database.

Supporting classes for the CustomAdapter class include:

- CustomEnvironment class that implements IAutomationEnvironment
- CustomAutomationClass that implements IAutomationClass
- CustomAutomationEventDescriptor class
- CustomAutomationMethodDescriptor class
- CustomAutomationPropertyDescriptor class

In addition to these classes, the ZIP file also includes the environment XML file and a utility class that parses that file.

You can also access a RemoteObject from the custom agent that provides user session information. To simplify, this example writes the data out to a trace log rather than to a database.

Using the CustomAdapter class

Follow the step-by-step instructions in this section to create a Flex application that records automation events by using the CustomAdapter class and its supporting files.

You cannot use this custom agent with agents that use different environment information, such as the QTP agent included in the Flex automation feature. So, if you use the agent in this example, do not use QTP to record scripts.

Set up your Flex installation

Before you can create an application that uses the automation API to record automatable events, you must do the following:

1 Make sure that you are running Flex Builder Professional.

2 Extract all the files in the customagent_src.zip file to a directory; for example, c:/myfiles/flex3/agent. This file contains the agent and helper classes in the custom.* package. It also includes MySQL and PHP code that you can use to connect your Flex application to a database.

3 Save the AutomationGenericEnv.xml file at the top level directory where you will put the Flex application. In this case, store it in c:/myfiles/flex3/agent.

After you set up your environment, you can edit the CustomAdapter class to record an interaction with a Flex application.

Edit the CustomAdapter class

The CustomAdapter class handles the `RECORD` events in the `recordHandler()` method. The `RECORD` event has the following properties:

- `name` — The agent's name of the event that triggered the call to the agent; for example, Click. This is the name as it is defined in the agent's environment.
- `replayableEvent` — The event that triggered the call to the agent.
- `automationObject` — The control that triggered the `replayableEvent`; for example, if the user clicked a button, this property contains a reference to the Button control. You can access the `automationObject` property by casting it to an IAutomationObject. You can then access properties of that object, such as a Button `label` property, by using the AutomationManager `getProperties()` method, as the following example shows:

```
var obj:IAutomationObject = event.automationObject;
var label:String = automationManager.getProperties(obj, ["label"])[0];
```

- `args` — Additional information about the event, such as the text entered in a TextInput control or whether the Alt key was held down during a mouse click. You can view all of the arguments by using code similar to the following:

```
for (var i:int = 0; i<event.args.length; i++) {
    trace("event.args[" + i + "]: " + event.args[i]);
}
```

The default version of the CustomAdapter `recordHandler()` method contains code that writes event information to a database. Follow the steps in this section to simplify that method so that you can run the example without configuring a database. After executing these steps, you can go back and create a database and revert the CustomAdapter class's `recordHandler()` method.

Simplify the CustomAdapter example

1 Open the CustomAdapter.as file.

2 Edit the `recordHandler()` method. Comment out the service access and replace it with trace statements; for example:

```
trace("automation name:" + obj.automationName);
trace("event name:" + event.name);
trace("replayable event:" + event.replayableEvent);
trace("event target:" + event.target);
// Arguments to be sent are '#' separated.
var arguments:String = event.args.join("#");
trace("args:" + arguments);

// Show all event args.
for (var i:int = 0; i<event.args.length; i++) {
    trace("event.args[" + i + "]: " + event.args[i]);
}
trace("------------------------");
```

You can later add the database support by uncommenting the service methods.

3 Edit the constructor. Add a `trace()` method such as the following:

```
trace("in CustomAdapter constructor")
```

This ensures that the custom agent is being instantiated when the application starts up.

4 Compile the custom agent's SWC file. To do this, you use the compc utility and include the classes in the custom package, as the following example shows:

```
compc -source-path+=c:/myfiles/flex3/agent -include-classes
    custom.CustomAdapter custom.CustomAutomationClass
    custom.CustomAutomationEventDescriptor
    custom.CustomAutomationMethodDescriptor
    custom.CustomAutomationPropertyDescriptor
    custom.CustomEnvironment
    custom.utilities.EnvXMLParser
    -library-path+=c:/home/dev/depot/flex/sdk/frameworks/libs
    -output=c:/myfiles/flex3/agent/CustomAgent.swc
```

This creates the CustomAgent.swc file in the c:/myfiles/flex3/agent directory.

Create and run the Flex application

When you use automation with Flex, you must create a Flex application and supporting wrapper files.

1 Create Main.mxml and store it in the same directory as the CustomAgent.swc file (for example, c:/myfiles/flex3/agent), as the following sample MXML file shows:

```
<?xml version="1.0"?>
<!-- agent/Main.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" height="100%" width="100%">
    <mx:Script><![CDATA[
        private function changeLabel(newLabel:String):void {
            b1.label = newLabel;
        }
    ]]></mx:Script>
    <mx:TextInput id="ti1" text=""/>
    <mx:Button id="b1" label="Change Label" click="changeLabel(ti1.text)" />
</mx:Application>
```

2 Compile the Flex application. You must include the CustomAgent.swc file as a library, as the following example shows:

```
mxmlc -include-libraries+=c:/myfiles/flex3/agent/CustomAgent.swc
c:/myfiles/flex3/agent/Main.mxml
```

This creates Main.swf in the c:/myfiles/flex3/agent directory.

3 Flex Builder creates an HTML wrapper for you. If you use the command-line compiler, you must create one manually. In the wrapper, you request the SWF file; if you try to run the SWF file directly, file security errors prevent you from loading the environment XML file. The following example wrapper loads the mysource.js file:

```
<html><body>
<script src="mysource.js"></script>
</body></html>
```

4 Create a JavaScript file that defines the `<OBJECT>` and `<EMBED>` tags that embed your Flex application's SWF file. For example:

```
document.write("<object id='tempId' classid='clsid:D27CDB6E-AE6D-11cf-96B8-
444553540000'  height='100%' width='100%'>");
document.write("<param name='src' value='Main.swf'/>");
document.write("<embed name='Main' src='Main.swf' height='100%' width='100%'/>");
document.write("</object>");
```

5 Copy the application SWF, wrapper, JavaScript, and environment XML files to a web server. The web server's directory now contains:

- Main.swf
- index-simple.html
- mysource.js
- AutomationGenericEnv.xml

The XML file is loaded at run time, so it must accessible by the SWF file on the web server.

6 Request the html wrapper from the web server; for example:

```
http://localhost:8100/flex/agent/index-simple.html
```

7 Run the application and change the name of the label to `42`. The trace log should contain something like the following:

```
in CustomAdapter constructor
automation name:ti1
event name:SelectText
event target:[object AutomationManager]
args:0#0
event.args[0]: 0
event.args[1]: 0
------------------------
automation name:ti1
event name:Input
event target:[object AutomationManager]
args:42
event.args[0]: 42
------------------------
automation name:Change Label
event name:Click
event target:[object AutomationManager]
args:
------------------------
```

Creating a replaying agent

You can create a custom agent that records and replays user interaction with a Flex application. This process is considerably more complex than simply recording user interaction for the purpose of metrics logging.

You cannot use this custom agent with agents that use different environment information, such as the QTP agent that is included with the Flex automation feature. Therefore, if you use this agent, do not use QTP to record and play back scripts.

Using the AutoQuick example

The AutoQuick example is available as a separate download (autoquick_src.zip file) at http://www.adobe.com/go/flex_automation_agent_apps.

Before you write a custom replaying agent, you should try the AutoQuick example, which shows you what type of information an agent must record so that it can play back the user interaction.

The AutoQuick example shows you how to specify codecs in the XML file and create a map of codecs that you can use for value translation.

1 Download and expand the autoquick_src.zip file: http://www.adobe.com/go/flex_automation_agent_apps.

2 Build the AutoQuick.swc file by using the instructions in "Building the AutoQuick.swc file" on page 394.

3 Copy the AutoQuick.swc file to a location that is accessible by your application's compiler.

4 Include the AutoQuick.swc libraryin your Flex application.

If you are using Flex Builder, add the AutoQuick.swc file by selecting Project > Properties > Flex Build Path > Library Path. Make sure that the Merged Into Code option is enabled.

If you are using the command-line compiler, add an entry in your flex-config.xml file that points to this SWC file; for example:

```
<include-libraries>
    <library>c:/myfiles/flex3/agent/simpler_replay/AutoQuick.swc</library>
</include-libraries>
```

Alternatively, on the command line, you can add the files by using the `include-libraries` option; for example:

```
mxmlc -C:\home\dev\depot\flex\sdk\bin>mxmlc -include-
libraries+=c:/myfiles/flex3/agent/simpler_replay/AutoQuick.swc
c:/myfiles/flex3/agent/myreplay/ReplayExample.mxml
```

Be sure to use the += operator to add the SWC file; this ensures that you do not replace other SWC files in your library path, but rather append this SWC file to the path.

5 Create a sample application that allows some kind of user interaction. For example, you could create a simple application that has just a Button control:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            private function clickHandler():void {
                b1.label = "clicked";
            }
        ]]>
    </mx:Script>
    <mx:Button id="b1" label="click me" click="clickHandler()"/>
</mx:Application>
```

6 Flex Builder creates an HTML wrapper for you. If you use the command-line compiler, you must create one manually. You can use the following sample wrapper:

```
<html lang="en">
 <head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
<title>ReplayExample</title>
 </head>
 <body>
<object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
 id="ReplayExample" width="100%" height="100%"
 codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab">
 <param name="movie" value="ReplayExample.swf" />
 <param name="quality" value="high" />
 <param name="allowScriptAccess" value="sameDomain" />
 <embed src="ReplayExample.swf" quality="high"
width="100%" height="100%" name="applicationReplayExample"
quality="high"
allowScriptAccess="sameDomain"
type="application/x-shockwave-flash"
pluginspage="http://www.macromedia.com/go/getflashplayer">
 </embed>
</object>
 </body>
</html>
```

7 Copy the wrapper and the application SWF file to your web server. You must deploy your sample application to a web server.

8 Copy the AutoQuickEnv.xml file from the autoquick_src.zip file to the same directory on your web server as your sample application SWF and wrapper files.

Now that you have created an application that uses the AutoQuick toolbar, you can record and play back events with it.

Record with the AutoQuick toolbar

1 Request the application SWF file. For example:

```
http://localhost:8100/replay/default.htm
```

2 The application opens with a Button control and the Automation pop-up window.

3 Click the Record button in the Automation pop-up window.

4 Click the button that you created in your application. The button's label should change from "click me" to "clicked".

5 Click the Stop button in the Automation pop-up window. The text area of the Automation pop-up window now contains the recorded interaction with the Flex application.

6 Copy the text from the Automation pop-up window to a text editor. It should look like the following:

```
<Records>
 <Step
    id="automationClassName{FlexApplication
        string}automationName{ReplayExample string}label{
        string}automationIndex{index:-1 string}id{ReplayExample string}
        className{ReplayExample string}|automationClassName{FlexButton
        string}automationName{click%20me string}label{click%20me
        string}automationIndex{index:0 string}id{b1 string}className
        {mx.controls.Button string}"
    method="Click">
<Args value=""/>
 </Step>
</Records>
```

The `<Records>` tag wraps the entire script. Each event in the script is represented by a `<Step>` tag. Within the step tag, `<Args>` child tags can describe arguments to the event. For example, if the event is a `MouseClick` event, the `<Args>` child tags define what keys were pressed at the time of the event.

The `id` attribute of the `<Step>` tag contains the entire control hierarchy for each target of an event. Each object in the `id` attribute is separated by a pipe (|), with the top-most object in the hierarchy occurring first. For example, in an application that has a TextInput control inside an HBox container, you would have an `id` attribute similar to the following:

```
Application | HBox | TextInput
```

Each object in the hierarchy is defined by a set of properties that use the following syntax:

property_name{*value type*}

The ReplayExample sample shows the following information about the application:

```
automationClassName{FlexApplication string}
automationName{ReplayExample string}
label{string}
automationIndex{index:-1 string}
id{ReplayExample string}
className{ReplayExample string}
```

It also shows the following information about the Button control, including the label:

```
automationClassName{FlexButton string}
automationName{click%20me string}
label{click%20me string}
automationIndex{index:0 string}
id{b1 string}
className{mx.controls.Button string}
```

The object definitions contain values for all of the attributes that are defined in the AutoQuickEnv.xml file for objects of that type. This ensures that the object can be uniquely identified in the script. This entry in the id attribute identifies the Button control as an object with the `automationClassName` property set to FlexButton, `automationName` set to label, label is "click me", `automationIndex` is 0, id is b1, and the canonical classname is `mx.controls.Button`.

The `method` attribute of the `<Step>` tag defines the event that was recorded.

Play back the recorded session with the AutoQuick toolbar

1. Restart your Flex application by either refreshing the browser or closing and reopening the browser.
2. Copy the recorded script from your text editor and paste it into the Automation pop-up window's text area.
3. Click the Play button. The Button control's label should change from "click me" to "clicked."

About the AutoQuick example

The AutoQuick example uses a variety of assets:

- AQToolBar.mxml — Defines the floating window that provides record, stop, and play functions. This component calls the methods of the helper classes.
- AQAdapter.as — Defines the agent class for the AutoQuick example. This class is a mixin, which triggers a call to its static `init()` method from the SystemManager class when the application starts up.
- AutoQuickEnv.xml — Defines what components and their methods, properties, and events can be recorded with the automation API. This is similar to the AutomationGenericEnv.xml file that is used by the CustomAgent example.
- codec.* class files — The classes in the codec package are used to convert ActionScript types to agent-specific types; you can think of these classes as providing serialization for the ActionScript types. For example, the AutoQuick example uses the DatePropertyDecode class to convert Date objects to an encoded form. The serialized object can then be loaded by the agent at run time.

Customizing the AutoQuick example

Two classes that you will probably customize are the AQToolBar.mxml file and the AQAdapter.as file.

Adding functionality to the AQToolBar.mxml file is just like editing any other custom Flex component. You can rebrand it, resize it, or change any of its properties to make it work within your application framework. For example, you could add an HTTPService call to a PHP page that logs the results of the recording. This would let you create a web-based, reusable event recording tool, similar to the CustomAgent example.

You can also customize the AQAdapter class; for example, you could add information to what is recorded. The `<Step>` and `<Args>` data is defined in the `recordHandler()` method of this class. You can edit this class to add properties. For example, the `<Args>` values are currently represented as numeric values. If the user holds the Shift key down while clicking a Button control, the agent records `<Args value="2"/>`. You could add logic to that class that includes human-readable values, such as `<Args value="ShiftKey"/>`.

After you have customized the AutoQuick example's source files, you must rebuild the SWC file. For more information, see "Building the AutoQuick.swc file" on page 394.

Building the AutoQuick.swc file

The autoquick_src.zip file contains all of the classes that you need to build the AutoQuick.swc file. When you first unzip the file, or if you make changes to the source files, you will need to build your own SWC file.

To build your own AutoQuick.swc file from the AutoQuick source files, compile the custom SWC file by using the following compc command:

```
C:\flex\sdk\bin>compc
    -source-path+=c:/myfiles/flex3/agent/replay/AutoQuick
```

```
-output c:/myfiles/flex3/agent/replay/AutoQuick.swc
-include-classes AQAdapter custom.CustomAutomationClass AQEnvironment
    custom.CustomAutomationEventDescriptor custom.CustomAutomationMethodDescriptor
    custom.CustomAutomationPropertyDescriptor custom.utilities.EnvXMLParser
    IAQCodecHelper IAQMethodDescriptor IAQPropertyDescriptor codec.ArrayPropertyCodec
    codec.AssetPropertyCodec codec.AutomationObjectPropertyCodec
    codec.ColorPropertyCodec codec.DatePropertyCodec codec.DateRangePropertyCodec
    codec.DateScrollDetailPropertyCodec codec.DefaultPropertyCodec codec.HitDataCodec
    codec.KeyCodePropertyCodec codec.KeyModifierPropertyCodec codec.ListDataObjectCodec
    codec.ScrollDetailPropertyCodec codec.ScrollDirectionPropertyCodec
    codec.TabObjectCodec codec.TriggerEventPropertyCodec
```

The exact file paths may be different on your computer.