

MX



macromedia[®]
FLEX

Developing Flex Applications

Trademarks

ActiveEdit, ActiveTest, Add Life to the Web, Afterburner, Aftershock, Andromedia, Allaire, Animation PowerPack, Aria, Attain, Authorware, Authorware Star, Backstage, Blue Sky Software, Blue Sky, Breeze, Bright Tiger, Captivate, Clustercats, ColdFusion, Contents Tab Composer, Contribute, Design In Motion, Director, Dream Templates, Dreamweaver, Drumbeat 2000, EDJE, EJIPT, Extreme 3D, Fireworks, Flash, FlashHelp, Flash Lite, FlashPaper, Flex, Flex Builder, Fontographer, FreeHand, Generator, Help To Source, HomeSite, Hotspot Studio, HTML Help Studio, JFusion, JRun, Kawa, Know Your Site, Knowledge Objects, Knowledge Stream, Knowledge Track, LikeMinds, Lingo, Live Effects, MacRecorder Logo and Design, Macromedia, Macromedia Action!, Macromedia Central, Macromedia Flash, Macromedia M Logo and Design, Macromedia Spectra, Macromedia xRes Logo and Design, MacroModel, Made with Macromedia, Made with Macromedia Logo and Design, MAGIC Logo and Design, Mediamaker, Movie Critic, Open Sesame!, RoboDemo, RoboEngine JFusion, RoboHelp, RoboHelp Office, RoboInfo, RoboInsight, RoboPDF, 1-Step RoboPDF, RoboFlash, RoboLinker, RoboScreenCapture, ReSize, Roundtrip, Roundtrip HTML, Shockwave, Sitespring, Smart Publishing Wizard, Software Video Camera, SoundEdit, Titlemaker, UltraDev, Web Design 101, what the web can be, WinHelp, WinHelp 2000, WinHelp BugHunter, WinHelp Find+, WinHelp Graphics Locator, WinHelp Hyperviewer, WinHelp Inspector, and Xtra are either registered trademarks or trademarks of Macromedia, Inc. and may be registered in the United States or in other jurisdictions including internationally. Other product names, logos, designs, titles, words, or phrases mentioned within this publication may be trademarks, service marks, or trade names of Macromedia, Inc. or other entities and may be registered in certain jurisdictions including internationally.

Third-Party Information

This guide contains links to third-party websites that are not under the control of Macromedia, and Macromedia is not responsible for the content on any linked site. If you access a third-party website mentioned in this guide, then you do so at your own risk. Macromedia provides these links only as a convenience, and the inclusion of the link does not imply that Macromedia endorses or accepts any responsibility for the content on those third-party sites.

Copyright © 2004 Macromedia, Inc. All rights reserved. This manual may not be copied, photocopied, reproduced, translated, or converted to any electronic or machine-readable form in whole or in part without prior written approval of Macromedia, Inc. Part Number ZFE15M100

Acknowledgments

Project Management: Stephen M. Gilson

Writing: Matthew J. Horn, Mike Peterson

Editing: Linda Adler

Production Management: Patrice O'Neill

Media Design and Production: Adam Barnett, John Francis

Second Edition: November 2004

Macromedia, Inc.
600 Townsend St.
San Francisco, CA 94103

CONTENTS

| | |
|---|----|
| INTRODUCTION: About Flex Documentation | 11 |
| Accessing the Flex documentation. | 12 |

PART I: Building User Interfaces to Flex Applications

| | |
|---|--------|
| CHAPTER 1: Using Flex Components | 15 |
| About components | 15 |
| Class hierarchy for components. | 16 |
| Using styles | 23 |
| Using behaviors. | 24 |
| Handling events | 25 |
| Applying skins. | 25 |
| Sizing components | 26 |
| Changing the appearance of a component at runtime | 28 |
| Extending components | 29 |
| CHAPTER 2: Using Controls | 31 |
| About controls. | 32 |
| Working with controls. | 35 |
| Button control. | 37 |
| CheckBox control | 40 |
| DateChooser control. | 42 |
| DateField control | 49 |
| HRule and VRule controls | 53 |
| HSlider and VSlider controls. | 56 |
| Label control. | 63 |
| Link control | 68 |
| Loader control. | 70 |
| NumericStepper control | 72 |
| ProgressBar control | 74 |
| RadioButton control | 77 |
| ScrollBar control | 81 |
| Text control. | 82 |
| TextArea control | 85 |
| TextInput control | 88 |

| | |
|---|---------|
| CHAPTER 3: Using Data Provider Controls | 91 |
| About data providers | 91 |
| List control | 105 |
| HorizontalList control | 116 |
| TileList control | 121 |
| DataGrid control | 127 |
| Tree control | 135 |
| ComboBox control | 141 |
| Menu control | 148 |
| MenuBar control | 153 |
| CHAPTER 4: Introducing Containers | 161 |
| About containers | 161 |
| Using containers | 162 |
| Controlling component sizing and positioning in a container | 170 |
| Using scroll bars | 180 |
| Creating component instances at runtime | 182 |
| Configuring containers | 185 |
| CHAPTER 5: Using the Application Container | 191 |
| Using the Application container | 191 |
| Application container syntax | 195 |
| Showing the download progress of an application | 200 |
| CHAPTER 6: Using Layout Containers | 205 |
| About layout containers | 206 |
| Canvas layout container | 206 |
| Box layout container | 209 |
| ControlBar layout container | 211 |
| DividedBox layout container | 212 |
| Form layout container | 215 |
| Grid layout container | 229 |
| Panel layout container | 234 |
| Tile layout container | 237 |
| TitleWindow layout container | 239 |
| CHAPTER 7: Using Navigator Containers | 247 |
| About navigator containers | 247 |
| ViewStack navigator container | 248 |
| LinkBar navigator container | 253 |
| TabBar navigator container | 255 |
| TabNavigator container | 259 |
| Accordion navigator container | 263 |

| | |
|---|-----|
| CHAPTER 8: Dynamically Repeating Controls and Containers | 269 |
| Using the Repeater object | 270 |
| Considerations when using a Repeater object | 279 |
| CHAPTER 9: Importing Images | 281 |
| About importing images | 281 |
| Importing images using the <mx:Image> tag | 282 |
| Controlling image importing with the <mx:Image> tag | 284 |
| Referencing images in other MXML tags | 292 |
| Importing images using Embed in ActionScript | 293 |
| CHAPTER 10: Importing Media | 297 |
| Using Media in Flex | 297 |
| Importing MP3 files | 298 |
| About the MediaDisplay control | 299 |
| About the MediaController control | 300 |
| About the MediaPlayer control | 302 |
| Sizing a media component | 302 |
| Adding a cue point | 302 |
| Syntax for the media controls | 303 |

PART II: Improving User Experience

| | |
|---|-----|
| CHAPTER 11: Building an Application with Multiple MXML Files | 309 |
| About MXML components | 309 |
| Creating MXML components | 311 |
| Passing component references | 316 |
| Using interfaces | 317 |
| CHAPTER 12: Working with ActionScript in Flex | 319 |
| Using ActionScript in Flex | 319 |
| Working with components | 320 |
| About scope | 325 |
| Using the doLater() method | 333 |
| CHAPTER 13: Using Events | 335 |
| About events | 335 |
| Handling events | 337 |
| Handling mouse events | 352 |
| Using base class events | 353 |

| | |
|--|-----|
| CHAPTER 14: Creating ActionScript Components | 359 |
| About ActionScript components | 359 |
| Defining custom user-interface components | 362 |
| Passing data to a custom tag | 362 |
| Defining events in ActionScript components | 363 |
| Using the createChildren() and createClassObject() methods | 367 |
| Using metadata keywords | 368 |
| Adding ActionScript components to the Flex environment | 373 |
| Defining nonvisual components | 375 |
| CHAPTER 15: Customizing Data Provider Controls | 379 |
| Creating a cell renderer | 379 |
| Creating a DataGrid header renderer | 384 |
| Displaying images in a List-based control | 386 |
| Validating data in a List-based control | 389 |
| Formatting data in a List-based control | 391 |
| CHAPTER 16: Using Styles and Fonts | 395 |
| About styles | 395 |
| Using external style sheets | 412 |
| Using local style definitions | 413 |
| Using the StyleManager | 417 |
| Using the setStyle() and getStyle() methods | 419 |
| Using inline styles | 421 |
| About fonts | 422 |
| CHAPTER 17: Using Themes and Skins | 429 |
| About skinning | 429 |
| About themes | 432 |
| Programmatically reskinning the CheckBox control | 433 |
| Programmatic skinning concepts | 438 |
| Graphical skinning | 449 |
| CHAPTER 18: Using Behaviors | 453 |
| Applying behaviors | 453 |
| Customizing an effect | 464 |
| Defining a custom effect | 468 |
| Defining and playing an effect in ActionScript | 471 |
| Using a custom effect trigger | 472 |
| CHAPTER 19: Creating Charts in Flex | 475 |
| About charting | 475 |
| Defining chart data | 483 |
| Creating charts | 487 |
| Formatting chart elements | 500 |
| Showing DataTips | 509 |

| | |
|---|---------|
| Using images in charts. | 512 |
| Using fills | 514 |
| Using Legend controls. | 521 |
| Stacking charts | 526 |
| Handling user interactions | 527 |
| Using effects with charts | 531 |
| Creating custom charts | 536 |
| CHAPTER 20: Using ToolTips | 539 |
| About ToolTips. | 539 |
| Creating ToolTips | 540 |
| Using the ToolTip Manager | 543 |
| CHAPTER 21: Using the Cursor Manager | 547 |
| About the Cursor Manager | 547 |
| Cursor Manager syntax | 551 |
| CHAPTER 22: Using the Drag and Drop Manager. | 553 |
| About the Drag and Drop Manager. | 553 |
| Using a List, Tree, or DataGrid control | 562 |
| Drag and Drop Manager syntax | 565 |
| CHAPTER 23: Using the History Manager | 569 |
| About history management | 569 |
| Using standard history management | 569 |
| Using custom history management | 571 |
| How the HistoryManager class saves and loads state | 574 |
| Using history management in a custom HTML file | 575 |
| CHAPTER 24: Improving Startup Performance. | 577 |
| About component instantiation. | 577 |
| Using the creationPolicy property | 578 |
| Manually instantiating controls. | 582 |
| Using the childDescriptors property | 584 |
| Setting container creation order. | 587 |
| CHAPTER 25: Using Runtime Shared Libraries. | 595 |
| About RSLs | 595 |
| Using RSLs | 597 |
| Creating RSLs for precompiled Flex applications. | 605 |
| RSL errors. | 606 |

| | |
|---|-----|
| CHAPTER 26: Printing from SWF Files | 607 |
| About Printing | 607 |
| Printing from the Flash Player context menu | 608 |
| Using the ActionScript PrintJob class | 609 |
| Starting a print job | 612 |

| | |
|---|-----|
| CHAPTER 27: Creating Accessible Applications | 617 |
| Accessibility overview | 617 |
| About screen reader technology | 619 |
| Configuring Flex applications for accessibility | 620 |
| Using accessible components and managers | 621 |
| Creating tab order and reading order | 623 |
| Accessibility for hearing-impaired users | 625 |
| Testing accessible content | 625 |

PART III: Data Access and Interconnectivity

| | |
|--|-----|
| CHAPTER 28: Managing Data in Flex | 629 |
| About Flex data management | 629 |
| Comparing Flex data management to other technologies | 633 |

| | |
|---|-----|
| CHAPTER 29: Binding and Storing Data in Flex | 637 |
| Binding data | 637 |
| Using data models | 647 |

| | |
|--|-----|
| CHAPTER 30: Using Data Services | 655 |
| About data services | 656 |
| Declaring a data service | 658 |
| Calling a data service | 659 |
| Handling data service results | 665 |
| Using a service with binding, validation, and event handlers | 669 |
| Handling asynchronous calls to data services | 670 |
| Using callback URLs | 672 |
| Generating debugging information for data services | 673 |
| Working with remote object services | 674 |
| Working with web services | 681 |
| Securing data services | 687 |
| Data service tag properties | 696 |
| Data service whitelist tags | 701 |

| | |
|--|-----|
| CHAPTER 31: Validating Data in Flex | 703 |
| Validating data | 703 |
| Using standard validators | 712 |

| | |
|------------------------------------|-----|
| CHAPTER 32: Formatting Data | 725 |
| Using formatters | 725 |
| Writing an error handler function | 726 |
| Using the standard formatters | 726 |
| Creating a custom formatter | 738 |

PART IV: Advanced Application Development and Debugging

| | |
|---|-----|
| CHAPTER 33: Debugging Flex Applications | 745 |
| About debugging | 745 |
| Enabling debug and warning messages | 746 |
| Using the error-reporting mechanism | 747 |
| Supported errors | 750 |
| About the debugger | 752 |
| Configuring the debugger | 754 |
| Invoking the debugger | 755 |
| Using the debugger | 757 |
| CHAPTER 34: Profiling ActionScript | 769 |
| About profiling | 769 |
| About the Profiler | 770 |
| Using the Profiler | 770 |
| Analyzing data | 776 |
| Troubleshooting | 780 |
| CHAPTER 35: Using the Flex JSP Tag Library | 781 |
| Introduction to the Flex JSP tag library | 781 |
| Using the Flex JSP tag library | 782 |
| About the Flex tags | 782 |
| Using the <mxml> tag | 785 |
| Using the <flash> tag | 788 |
| Using the <param> tag | 789 |
| Using the <flashvar> tag | 789 |

PART V: Administrating Applications

| | |
|---------------------------------------|-----|
| CHAPTER 36: Administering Flex | 795 |
| Overview | 795 |
| Using the command-line compiler | 796 |
| Using SWC files | 800 |
| Editing the flex-config.xml file | 806 |
| Changing application server settings | 817 |
| Configuring logging | 818 |

| | |
|---|-----|
| CHAPTER 37: Applying Flex Security | 823 |
| Flex security features | 823 |
| Flash Player security features | 825 |
| Security concerns of an open format technology | 832 |
| Resources | 833 |
| CHAPTER 38: Deploying Applications | 835 |
| About deploying | 835 |
| Adding Flex to your application server. | 836 |
| Distributing components. | 840 |
| Working with Flex applications | 842 |
| About the HTML wrapper | 845 |
| Flash Player detection and deployment | 853 |
| Managing Flash Player auto-update. | 857 |
| Passing request data to Flex applications | 859 |
| PART VI: Custom Components | |
| CHAPTER 39: Using Custom Flex 1.0 Skins and Components in Flex 1.5 | 863 |
| Overview. | 863 |
| Skinning in Flash. | 863 |
| Creating custom components in Flash. | 864 |
| Updating existing components | 864 |
| INDEX | 867 |

INTRODUCTION

About Flex Documentation

Developing Flex Applications provides the tools for you to develop Internet applications using Macromedia Flex. This book is intended for application developers who are learning Flex or wish to extend their Flex programming knowledge. It provides a solid grounding in the tools that Flex provides to develop applications.

Contents

| | |
|--|----|
| Using this manual | 11 |
| Accessing the Flex documentation | 12 |

Using this manual

This manual can help anyone who is developing Flex applications. However, this book is most useful if you have basic experience using Flex, or have read *Getting Started with Flex*. *Getting Started with Flex* provides an introduction to Flex and helps you develop the basic knowledge that makes using this manual easier.

Developing Flex Applications is divided into the following parts:

| Part | Description |
|---|---|
| Part I, "Building User Interfaces to Flex Applications" | Describes how to use Flex components to build the user interface to your application. |
| Part II, "Improving User Experience" | Describes how to improve the user experience by adding additional functionality to your application. |
| Part III, "Data Access and Interconnectivity" | Describes how to use Flex data models and data services. |
| Part IV, "Advanced Application Development and Debugging" | Describes how to debug and profile your Flex applications, add MXML code to your JSP pages, and create custom HTML wrappers for your Flex applications. |
| Part V, "Administrating Applications" | Describes how to manage and deploy applications. |
| Part VI, "Custom Components" | Describes how to create custom components for Flex using Macromedia Flash MX 2004. |

Accessing the Flex documentation

The Flex documentation is designed to provide support for the complete spectrum of participants.

Documentation set

The Flex documentation set includes the following titles:

| Book | Description |
|---|---|
| <i>Developing Flex Applications</i> | Describes how to develop your dynamic web applications. |
| <i>Getting Started with Flex</i> | Contains an overview of Flex features and application development procedures. |
| <i>Flex ActionScript and MXML API Reference</i> | Provides descriptions, syntax, usage, and code examples for the FlexAPI. |

Viewing online documentation

All Flex documentation is available online in HTML and Adobe Acrobat Portable Document Format (PDF) files. Go to the documentation home page for Flex on the Macromedia website: www.macromedia.com.

PART I

Building User Interfaces to Flex Applications

This part describes how to use Macromedia Flex components to build the user interface to your application.

The following chapters are included:

| | |
|--|-----|
| Chapter 1: Using Flex Components | 15 |
| Chapter 2: Using Controls | 31 |
| Chapter 3: Using Data Provider Controls | 91 |
| Chapter 4: Introducing Containers | 161 |
| Chapter 5: Using the Application Container | 191 |
| Chapter 6: Using Layout Containers | 205 |
| Chapter 7: Using Navigator Containers | 247 |
| Chapter 8: Dynamically Repeating Controls and Containers | 269 |
| Chapter 9: Importing Images | 281 |
| Chapter 10: Importing Media | 297 |

CHAPTER 1

Using Flex Components

Macromedia Flex provides a component-based development environment that you use to develop your applications. Components have a flexible set of characteristics that let you control and configure them as necessary for your application requirements. This chapter contains an overview of components, component syntax, and component configuration.

Contents

| | |
|---|----|
| About components. | 15 |
| Class hierarchy for components | 16 |
| Using styles | 23 |
| Using behaviors | 24 |
| Handling events. | 25 |
| Applying skins | 25 |
| Sizing components. | 26 |
| Changing the appearance of a component at runtime | 28 |
| Extending components | 29 |

About components

Flex includes a component-based development model that you use to develop your application and its user interface. You can use the prebuilt components included with Flex, you can extend components to add new properties and methods, and you can create new components as required by your application.

Components are extremely flexible and provide you with a great amount of control over the component's appearance, how the component reacts to user interactions, the font and font size of any text included in the component, the size of the component in the application, and many other characteristics.

This chapter contains an overview of many of the characteristics of components, including the following:

Styles Characteristics, such as font, font size, and text alignment. These are the same styles that you define and use with Cascading Style Sheets (CSS).

Behaviors Visible or audible changes to the component triggered by an application or user action. Examples of behaviors are moving or resizing a component based on a mouse click.

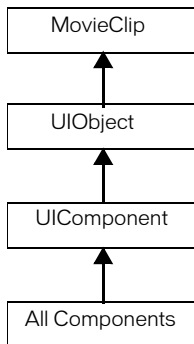
Events Application or user actions that require a component response. Events include component creation, mouse actions such as a mouse over, and button clicks.

Skins Symbols that control a component's appearance.

Size Height and width of a component. All components have a default size. You can use the default size, specify your own size, or let Flex resize a component as part of laying out your application.

Class hierarchy for components

Flex components are implemented as a class hierarchy in ActionScript. Each component in your application is therefore an instance of an ActionScript class. The following figure shows this hierarchy:



All components are derived from the ActionScript classes `MovieClip`, `UIObject`, and `UIComponent` and inherit the properties and methods of their parent classes. In addition, components inherit other characteristics of the parent classes, including event, style, and behavior definitions.

In general, you can do the following with components:

- Set most writable properties, and some read-only properties, of an ActionScript class in MXML.
- Set all event, style, and behavior properties of an ActionScript class in MXML.

Note: You cannot reference ActionScript methods in MXML; you can only reference them in ActionScript.

Using the MovieClip class

The MovieClip class is the base class for all Flex components. The MovieClip class is part of the ActionScript language; it is not specific to Flex. Therefore, the documentation on the MovieClip API is contained in *Flex ActionScript Language Reference*.

The following table lists the properties and methods inherited by all components from the MovieClip parent class that Macromedia recommends that you use with Flex. This table also lists properties and methods that you should not use with Flex, and the Flex substitute. Do not use any properties or methods of the MovieClip class that are not in this table.

| MovieClip | Property/Method | Notes |
|-----------------|------------------------|---|
| General Methods | attachAudio() | |
| | attachMovie() | Use <code>UIObject.createObject()</code> and <code>View.createChild()</code> instead. |
| | createEmptyMovieClip() | Use <code>UIObject.createEmptyObject()</code> instead. |
| | createTextField() | Use <code>UIObject.createLabel()</code> instead. |
| | duplicateMovieClip() | Use <code>createObject()</code> and <code>createChild()</code> instead. |
| | getBounds() | |
| | getBytesLoaded() | Use the Loader control instead. |
| | getBytesTotal() | Use the Loader control instead. |
| | getDepth() | You can use this method, but you should use the DepthManager instead. |
| | getInstanceAtDepth() | You can use this method, but you should use the DepthManager instead. |
| | getNextHighestDepth() | You can use this method, but you should use the DepthManager instead. |
| | getSWFVersion() | |
| | getTextSnapshot() | |
| | getURL() | |
| | globalToLocal() | |
| | hitTest() | |
| | loadMovie() | Use the Loader control instead. |
| | loadVariables() | |
| | localToGlobal() | |
| | removeMovieClip() | Use <code>destroyObject()</code> instead. |
| | setMask() | Use <code>UIObject.setMask()</code> instead. |
| | startDrag()/stopDrag() | Do not use in Flex. Use the Drag Manager instead. |
| | unloadMovie() | |

| MovieClip | Property/Method | Notes |
|-----------------|---|---|
| Drawing methods | beginFill() beginGradientFill() clear() curveTo() endFill() lineStyle() lineTo() moveTo() | |
| Properties | _alpha _droptarget enabled focusEnabled _focusrect hitArea menu _name _soundbuftime tabChildren tabEnabled tabIndex _target _url | Use <code>UIObject.alpha</code> instead. You should use the <code>DepthManager</code> instead. |

Using the UIObject and UIComponent classes

The `UIObject` and `UIComponent` classes are specific to Flex, and the documentation on their API is in *Flex ActionScript and MXML API Reference*. The following table lists the optional, common properties of components that extend the `UIObject` and `UIComponent` classes:

| Property | Type | Description |
|----------------------|-------------------------|---|
| <code>alpha</code> | Number | Specifies the transparency of a component. Possible values are 0 (invisible) through 100 (opaque). Device fonts do not honor the <code>alpha</code> setting, but embedded fonts do. The default value is 100. |
| <code>enabled</code> | Boolean | Setting to <code>true</code> allows the component to accept keyboard focus and mouse input. The default value is <code>true</code> . If you set <code>enabled</code> to <code>false</code> for a container, Flex dims the color of the container and of all of its children, and blocks user input to the container and to all of its children. |
| <code>height</code> | Number or String* | Sets the height of the component, in pixels or in percentage of available space (100% is the full span of the parent container or, for <code><mx:Application></code> tags, the full span of the browser). Specifying a percentage value makes the component resizable. Input values can either be a number or a string consisting of a number followed by a percent sign (%). Output values are in pixels regardless of how the property was set. |
| <code>id</code> | String | Specifies the component identifier; this value becomes the instance name of the object. |

| Property | Type | Description |
|--------------------------------|---------|---|
| <code>maxHeight</code> | Number | Number that specifies the maximum height of the component, in pixels. |
| <code>maxWidth</code> | Number | Number that specifies the maximum width of the component, in pixels. |
| <code>minHeight</code> | Number | Number that specifies the minimum height of the component, in pixels. |
| <code>minWidth</code> | Number | Number that specifies the minimum width of the component, in pixels. |
| <code>mouseX</code> | Number | Read-only property that contains the x-position, in pixels, of the mouse pointer relative to the upper-left corner of the component. |
| <code>mouseY</code> | Number | Read-only property that contains the y-position, in pixels, of the mouse pointer relative to the upper-left corner of the component. |
| <code>parentApplication</code> | Object | Contains a reference to the Application object for the application that contains the component. The value is undefined for the top-level Application object. |
| <code>parentDocument</code> | Object | Contains a reference to the next level up in the document chain of a Flex application. |
| <code>percentHeight</code> | Number | Number that specifies the last declared variable height percentage. This read-only property returns undefined if a percent-based height has never been set. |
| <code>percentWidth</code> | Number | Number that specifies the last declared variable width percentage. This read-only property returns undefined if a percent-based width has never been set. |
| <code>preferredHeight</code> | Number | Number that specifies the preferred height of the component, in pixels. This is the height used for the object if no height is specified. Set to 0 when <code>height</code> is set to a percentage value. |
| <code>preferredWidth</code> | Number | Number that specifies the preferred width of the component, in pixels. This is the width used for the object if no width is specified. Set to 0 when <code>width</code> is set to a percentage value. |
| <code>scaleX</code> | Number | Number that specifies the horizontal scaling percentage. The default value is 100. |
| <code>scaleY</code> | Number | Number that specifies the vertical scaling percentage. The default value is 100. |
| <code>styleName</code> | String | Specifies the style class to apply to the component. |
| <code>tabEnabled</code> | Boolean | Specifies whether the component is included in automatic tab ordering (<code>true</code>) or not (<code>false</code>). The default value is <code>true</code> . |
| <code>tabIndex</code> | Number | Number specifying the component's tabbing order in relation to other components in an application. |
| <code>toolTip</code> | String | Text string displayed when the mouse pointer hovers over that component. |
| <code>visible</code> | Boolean | Boolean value that specifies whether the container is visible or invisible. The default value is <code>true</code> to specify visible. |

| Property | Type | Description |
|----------|-------------------------|--|
| width | Number or String* | Specifies the width of the component, in pixels or in percentage of available space (100% is the full span of the parent container or, for <mx:Application> tags, the full span of the browser). Specifying a percentage value makes the component resizable. Input values can either be a number or a string consisting of a number followed by a percent sign (%). Output values are in pixels regardless of how the property was set. |
| x | Number | Number that specifies the component's x-position within its parent container. Only recognized when the component is a child of a Canvas container. |
| y | Number | Number that specifies the component's y-position within its parent container. Only recognized when the component is a child of a Canvas container. |

* These properties are untyped but only accept numbers and percents (expressed as strings) as input. Both return numbers on output.

Using MXML and ActionScript in an application

Every Flex component has an MXML API and an ActionScript API. A component's MXML tag properties are equivalent to its ActionScript properties, styles, behaviors, events, and skins. You can use both MXML and ActionScript when working with components.

To configure a component:

1. Set the value of a component property, event, style, or behavior declaratively in an MXML tag, or at runtime in ActionScript code.
2. Call a component's methods at runtime in ActionScript code. The methods of an ActionScript class are not exposed in the MXML API.

The following example creates a Button control in MXML. Clicking the Button control updates the text of a TextArea control using an ActionScript function.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

    <!-- ActionScript to handle event. -->
    <mx:Script>
        <![CDATA[
            function handleClick()
            {
                text1.text="Thanks for the click!";
            }
        ]]>
    </mx:Script>

    <!-- MXML component definition. -->
    <mx:Button id="button1" label="Click here!" width="100"
        click="handleClick()" fontSize="12" />

```

```
<mx:TextArea id="text1" />

</mx:Application>
```

where:

- `id` is a property inherited by the Button control from the UIObject class that you use to specify a unique identifier for the component. This property is optional, unless you want to process the component using ActionScript. The identifier becomes the symbol name of the component in ActionScript.
- `label` is a property defined by the Button control that specifies the text that appears in the button.
- `width` is a property inherited from the UIObject class that optionally specifies the width of the button, in pixels.
- `click` is an event defined by the Button control that specifies the ActionScript code executed when a user selects the Button control.
- `fontSize` is a style inherited from the UIObject class that specifies the font size of the label text, in pixels.

The `click` property can also take the ActionScript code as its value, without you having to specify it in a function. Therefore, you can rewrite this example as the following code shows:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

    <mx:Button id="button1" label="Click here!" width="100"
        click='text1.text="Thanks for the click!";' fontSize="12" />

    <mx:TextArea id="text1" />

</mx:Application>
```

While you can specify many statements of ActionScript code, separated by semicolons, as the value of the `click` property, for readability you should limit it to only one or two lines of code.

Accessing read-only ActionScript properties in MXML

ActionScript classes define some properties as read-write and others as read-only. To set the value of a read-write property, assign a value to the property in MXML or ActionScript. The previous example set the read-write `label` property of a Button control.

MXML lets you set some read-only properties of a class. For example, in MXML, you can set the `contentPath` property of the MediaPlayer control when you create it, as the following example shows:

```
<mx:MediaPlayer contentPath="http://myhost.com/media/MyVideo.flv" />
```

Initializing components at runtime

Flex evaluates MXML properties at compile-time to initialize your components. However, you might want to use some logic to determine initial values at runtime. For example, you might want to initialize a component with the current date or time. Flex must calculate this type of information when the application executes.

Every component supports the `initialize` event, which lets you specify `ActionScript` to initialize a component after the application loads, but before Flex renders it in Macromedia Flash Player. To initialize a component with `ActionScript`, set the `initialize` property to point to a function that you create in an `<mx:Script>` block.

The following example configures Flex to call the `initDate()` function when it initializes the `Label` control. When Flex finishes instantiating the `Label` control, and before it draws the first screen in the application, Flex calls the `initDate()` function.

```
<mx:Script>
    <![CDATA[
        function initDate() {
            label1.text = label1.text + Date();
        }
    ]]>
</mx:Script>

<mx:Label id="label1" text = "Today's Date: " initialize="initDate()"
    width="300" />
```

You can also express the previous example without a function call by adding the `ActionScript` code in the component's definition. The following example does the same thing, but without an explicit function call:

```
<mx:Label id="label1" text = "Today's Date: "
    initialize="label1.text = label1.text + Date();" width="300" />
```

As with other calls that are embedded within component definitions, you can add multiple `ActionScript` statements to the `initialize` property by separating each function or method call with a semicolon. The following example calls both the `initDate()` and the `setColor()` functions when the `label1` component is instantiated:

```
<mx:Label id="label1" text = "Today's Date: "
    initialize="initDate(); setColor('blue');" width="300" />
```

MXML and `ActionScript` summary

The following table summarizes the `MXML` and `ActionScript` component APIs that you use to configure components:

| | MXML example | ActionScript example |
|---------------------|--|--|
| Read-write property | <pre><mx:Tile id="tile1" label="My Tile" visible="true" /></pre> | <pre>tile1.visible="false"; tile1.label="My Tile";</pre> |
| Read-only property | For read-only properties that are settable in <code>MXML</code> . <pre><mx:MediaPlayback id="mp1" contentPath="http:// myhost.com/media/MyVideo.flv" /></pre> | Get the value of a read-only property: <pre>var cPath:String=mp1.contentPath;</pre> |
| Method | Methods are not available in <code>MXML</code> . | <pre>myList.sortItemsBy("data", "DESC");</pre> |

| | MXML example | ActionScript example |
|----------|--|---|
| Event | <pre><mx:Accordion id="myAcc" ... change="handleEvent()" /></pre> | <pre>listener = new Object(); listener.click = function(evtObj) { // Handler definition. } myButton.addEventListener("click", listener);</pre> |
| Style | <pre><mx:Tile id="tile1" marginTop="12" marginBottom="12" /></pre> | <pre>var currentTopM:Number = tile1.getStyle("marginTop"); tile1.setStyle("marginTop","12"); tile1.setStyle("marginBottom","12");</pre> |
| Behavior | <pre><mx:Tile id="tile1" showEffect="WipeRight" /></pre> | <pre>var currentTopM:String = tile1.getStyle("showEffect");</pre> <p>Use the <code>setStyle()</code> method to set a behavior, as shown in the Style row of this table.</p> |

Using styles

Flex defines styles for setting some of the characteristics of components, such as fonts, margins, and alignment. These are the same styles as defined and used with Cascading Style Sheets (CSS). The `UIObject` and `UIComponent` classes define the global styles available for all components. In addition, each component can define its own styles.

You can set all styles in MXML as tag properties. Therefore, you can set the margins for a `Box` container using the following syntax:

```
<mx:VBox id="myVBox1" marginTop="12" marginBottom="12" />
  <mx:Button label="Submit"/>
</mx:VBox>
```

You can also configure styles in MXML using the `<mx:Style>` tag, or in ActionScript using the `setStyle()` method. The `<mx:Style>` tag contains style declarations using CSS syntax or a reference to an external file that contains style declarations, as the following example shows:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">

  <mx:Style>
    .myClass { horizontalAlign: "left";}
    Box { marginTop: "12"; marginBottom: "12";}
    Tile { verticalGap: "10"; horizontalGap: "10";}
  </mx:Style>

  <mx:VBox id="myVBox1" />
    <mx:Button label="Submit"/>
  </mx:VBox>

  <mx:VBox id="myVBox2" styleName="myClass" />
    <mx:Button label="Submit"/>
  </mx:VBox>

</mx:Application>
```

A *type selector* applies a style to all instances of a particular component type. In the preceding example, you define the top and bottom margins for all Box containers, and the gap for all Tile containers. In some cases, the style is inherited by the component's children.

A *class selector* in a style definition, defined as a label preceded by a period, defines a new named style, such as `myClass` in the preceding example. After you define it, you can apply the style to any component using the `styleName` property. In the preceding example, you apply the style to the second VBox container.

You can set a style in ActionScript using the `setStyle()` method. The `setStyle()` method takes two arguments: the style name and the value. For example, you could set the margin as the following example shows:

```
myVBox1.setStyle("marginRight", "12");  
myVBox1.setStyle("marginLeft", "12");
```

For more information on styles, see [Chapter 16, “Using Styles and Fonts,” on page 395](#).

Using behaviors

A *behavior* is a combination of a trigger paired with an effect. A *trigger* is an action, much like an event, such as a mouse click on a component, a component getting focus, or a component becoming visible. An *effect* is a visible or audible change to the component occurring over a period of time, measured in milliseconds. Examples of effects are fading, resizing, or moving a component. You can define multiple effects for a single trigger.

Behaviors let you add animation, motion, and sound to your application in response to some user or programmatic action. For example, you can use behaviors to cause a dialog box to bounce slightly when it receives focus, or to play a sound when the user enters an invalid value.

Flex uses a Cascading Style Sheet (CSS) to define behavior properties. You can reference the style properties as tag properties in MXML, within the `<mx:Style>` tag, or in an ActionScript function. For example, to configure the effect for the show trigger within an `<mx:Image>` tag, you use the following MXML syntax:

```
<mx:Image showEffect="Fade" source="first.jpeg" />
```

In this example, the Image control fades in over 500 milliseconds, the default time interval for a Fade effect.

For more information on behaviors, see [Chapter 18, “Using Behaviors,” on page 453](#).

Handling events

Flex applications are event-driven. *Events* let a programmer know when the user has interacted with an interface component, and also when important changes have happened in the appearance or life cycle of a component, such as the creation or destruction of a component or its resizing.

When an instance of a component raises an event, two groups of objects are notified, in the following order:

1. The component instance
2. Objects that have registered as listeners for that event

You define event handlers in ActionScript to process events. You register event handlers for events either in the MXML declaration for the component or in ActionScript. The following example registers an event handler in MXML for an Accordion container `change` event:

```
<mx:Script>
  <![CDATA[
    function handleAccChange()
    {
      trace(myAcc.selectedChild.label);
    }
  ]]>
</mx:Script>

<mx:Accordion id="myAcc" change="handleAccChange()"/>
```

You can also pass an event object from the component to the event handler. Depending on the type of event, the event object can contain additional information that you can use to handle the event. For more information on events, see [Chapter 12, “Working with ActionScript in Flex,” on page 319](#).

Applying skins

Skins are symbols a component uses to display its appearance. Flex components can be composed of one or more symbols that make up their skins. For example, the down arrow of the `ScrollBar` component is made up of three skins: `ScrollDownArrowDisabled`, `ScrollDownArrowUp`, and `ScrollDownArrowDown`. Some components share skins. Components that use scroll bars—including all containers—share the skins with the `ScrollBar` component.

You can edit existing skins and create new skins to change the appearance of a component. There are two ways to reskin Flex components: graphically and programmatically. To graphically reskin components, you edit sample theme FLA files and redraw a component’s skin. To programmatically reskin Flex components, you edit ActionScript class files that define the skin. In both cases, you export the new skins as SWC files and use them in your Flex applications as themes.

Not all components can be reskinned both programmatically and graphically, and some components cannot be reskinned at all. For more information on skinning, see [Chapter 17, “Using Themes and Skins,” on page 429](#).

Sizing components

Each component defines rules for determining its size in a Flex application when no size is explicitly specified. For example, a Button control sizes itself to fit its label text, while an Image control sizes itself to the size of the imported image. In Flex, the default size of a component is called its *preferred* size and each component has a preferred height and a preferred width.

The preferred size of a component is not necessarily a fixed value. For a Button control, the preferred size is a size large enough to fit its text label. For containers, the preferred width is sometimes determined by the components within the container. For example, the preferred width of a vertical Box (VBox) container is the preferred width of its largest child component, plus any margins.

At runtime, Flex checks whether your components specify width and height. If they don't, Flex calculates the preferred size of each component and draws your application accordingly. If they specify fixed widths and heights, Flex applies those sizes, and then performs any calculations needed to determine the preferred size of other components. If you have any flexible components, Flex determines their size by the size of their containers.

You can freely mix fixed-size components and flexible, or resizable, components, as long as the sum of the percentages of all flexible components remains less than or equal to 100%. The sum of the percentages of all flexible components can be as large as 100% even if you also use fixed-size components, which means that components can request space that isn't actually available. Each flexible component strives to match the specified percentage of its parent container's size, however, the percentages scale if not all of the space is available.

Note: There may be some cases where it is necessary to allow the sum of all percentages to exceed 100% in order to achieve the desired behavior. However, Macromedia recommends observing this limit whenever possible.

Allocating space to components

Flex allocates the available space within a container to its child components, as follows:

1. Space is given to fixed-size components.
2. The remaining space is divided among the resizable components, while respecting the percentage values you specify. Flex attempts to match the size of each component to the percentage of its container that is specified in its definition. When there is not enough space to give each component its requested percentage, Flex evenly scales all percentages and maintains the ratio between specified percentage-based sizes.
3. If the requested percentage would result in a component being sized smaller than its `minHeight` or `minWidth` value, or larger than its `maxHeight` or `maxWidth` value, the size is set to the minimum or maximum value and the component is treated like a fixed-size component for the rest of the sizing process. The calculation to determine the size of the flexible components, begun in step 1, is restarted at this point.
4. If the space consumed by all components exceeds the size of the container, scroll bars appear. This can happen only if all components are a fixed size (or resized to be so because of minimum size constraints).

Examples of component sizing

The following example shows sizing within a fixed-width container when some of its child controls are specified with fixed widths and some with flexible widths:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:HBox width="250">
    <mx:Label text="Hello" width="50"/>
    <mx:Image source="@Embed(logo.gif)" width="75%"/>
    <mx:Button label="Button" width="25%" />
  </mx:HBox>
</mx:Application>
```

The application includes a 50-pixel-wide label, a 75% width image, and a 25% width button inside of a 250-pixel-wide HBox. Flex includes an 8-pixel gap between components by default, so the HBox has 184 pixels free for the two flexible components. The image takes up 138 pixels (60% of the container) and the button is 46 pixels wide (20%). This maintains the 3-to-1 ratio of the specified percentages, while respecting the absolute size requirements of the label control and the gaps between components.

In other words, the preceding example yields exactly the same results as this example:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:HBox width="250">
    <mx:Label text="Hello" width="50"/>
    <mx:Image source="@Embed(logo.gif)" width="138"/>
    <mx:Button label="Button" width="46" />
  </mx:HBox>
</mx:Application>
```

Flexible sizing doesn't provide much benefit until your containers also resize. The only tangible benefit is that it removes the need to explicitly consider the gaps and margins of a container in your calculations. In the following example, the series of controls on the left resize as you resize your browser, but the same controls on the right side remain a fixed size because their container is fixed:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:HBox width="100%">
    <mx:HBox width="40%">
      <mx:Label text="Hello" width="50"/>
      <mx:Image source="@Embed(logo.gif)" width="75%"/>
      <mx:Button label="Button" width="25%" />
    </mx:HBox>
    <mx:HBox width="250">
      <mx:Label text="Hello" width="50"/>
      <mx:Image source="@Embed(logo.gif)" width="75%"/>
      <mx:Button label="Button" width="25%" />
    </mx:HBox>
  </mx:HBox>
</mx:Application>
```

For more information on sizing components, see [Chapter 4, “Introducing Containers,”](#) on [page 161](#).

Changing the appearance of a component at runtime

You can modify the look, size, or position of a component at runtime using the following component properties or ActionScript methods:

- `x` and `y`
- `width` and `height`
- `setStyle(styleName, value)`

You can only set the `x` and `y` properties of a component when the component is within a Canvas container. All other containers perform automatic layout to set the `x` and `y` properties of their children using layout rules.

For example, you could use the `x` and `y` properties to reposition a Button control 10 pixels to the right and 10 pixels down in response to a Button click, as the following example shows:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">

    <mx:Script>
        <![CDATA[
            function moveButton()
            {
                myButton.x = myButton.x +10;
                myButton.y = myButton.y +10;
            }
        ]]>
    </mx:Script>

    <mx:Canvas>
        <mx:Button id="myButton" label="Move Button" click="moveButton()"/>
    </mx:Canvas>

</mx:Application>
```

In this application, you can move the Button control without concern for other components. However, moving a component in an application that contains multiple components, or modifying one child of a container that contains multiple children, can cause one component to overlap another, or in some other way affect the layout of the application. Therefore, you should be careful when you perform runtime modifications to container layout.

You can set the `width` and `height` properties for a component regardless of the container that holds it. The following example increases the `width` and `height` of a Button control by 10 pixels each time the user selects it:

```
<mx:Script>
    <![CDATA[

        function resizeButton()
        {
            myOtherButton.height = myOtherButton.height + 10;
            myOtherButton.width = myOtherButton.width + 10;
        }
    ]]>
```

```

</mx:Script>

<mx:VBox borderStyle="solid" height="200" width="200" >

    <mx:Button id="myOtherButton" label="Resize Button" click="resizeButton()"/>

</mx:VBox>

```

If the container that holds the Button control is any container other than the Canvas container, it repositions its children based on the new size of the Button control. The Canvas container performs no automatic layout, so changing the size of one of its children does not cause the position or size of any other children within it to be modified.

Note: The stored values of width and height are always in pixels regardless of whether the values were originally set as fixed values, as percentages, or not set at all.

Extending components

Flex provides several ways for you to extend existing components or to create new components. By extending a component, you can add new properties or methods to it.

For example, the following MXML component, defined in the file MyComboBox.mxml, extends the standard ComboBox control to initialize it with the postal abbreviations of all 50 states:

```

<!-- MyComboBox.mxml -->
<?xml version="1.0"?>
<mx:ComboBox xmlns:mx="http://www.macromedia.com/2003/mxml">
    <mx:dataProvider>
        <mx:Array>
            <mx:String>AK</mx:String>
            <mx:String>AL</mx:String>
            ...
        </mx:Array>
    </mx:dataProvider>
</mx:ComboBox>

```

After you create it, you can use your new component anywhere in a Flex application by specifying its filename as its MXML tag name, as the following example shows:

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    xmlns:local="*">

    <local:MyComboBox id="stateNames" />

</mx:Application>

```

In this example, the new component is in the same directories as your application file, and maps the local namespace (*) local.

Flex provides several methods for creating custom components. The method you choose depends on your application and the requirements of your component. To create new components, use one of the following methods:

- Create components as MXML files and use them as custom tags in other MXML files. MXML components provide an easy way to extend an existing Flex component and encapsulate the appearance and behavior of a component in a custom MXML tag. For more information, see [Chapter 11, “Building an Application with Multiple MXML Files,” on page 309](#).
- Create components as ActionScript files and use them as custom tags. You can derive your components from the same class hierarchy as the Flex components. For more information, see [Chapter 14, “Creating ActionScript Components,” on page 359](#).

CHAPTER 2

Using Controls

Controls are user-interface components such as Button, TextArea, and ComboBox controls. This chapter describes how to use controls in a Macromedia Flex application.

Flex has two types of controls: basic and data provider. This chapter contains an overview of all Flex controls, and describes the basic Flex controls. For information on data provider controls, see Chapter 3, “Using Data Provider Controls,” on page 91.

Contents

| | |
|--|----|
| About controls | 32 |
| Working with controls | 35 |
| Button control | 37 |
| CheckBox control | 40 |
| DateChooser control | 42 |
| DateField control | 49 |
| HRule and VRule controls | 53 |
| HSlider and VSlider controls | 56 |
| Label control | 63 |
| Link control | 68 |
| Loader control | 70 |
| NumericStepper control | 72 |
| ProgressBar control | 74 |
| RadioButton control | 77 |
| ScrollBar control | 81 |
| Text control | 82 |
| TextArea control | 85 |
| TextInput control | 88 |

About controls

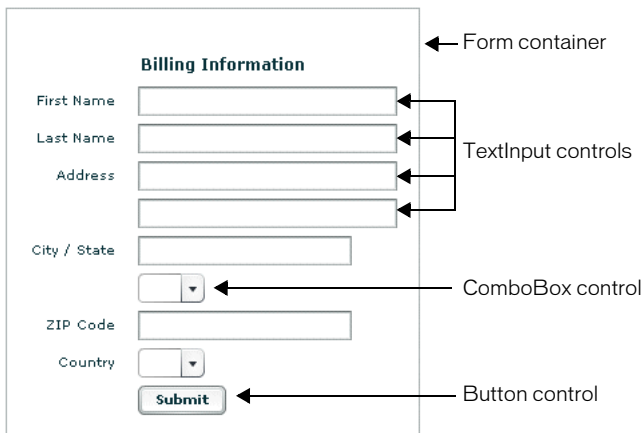
Controls are user-interface components, such as Button, TextArea, and ComboBox controls. You place controls in containers, which are user-interface components that provide a hierarchical structure for controls and other containers. Typically, you define a container, and then insert controls or other containers in it.

At the root of a Flex application is the `<mx:Application>` tag, which represents a base container that covers the entire Macromedia Flash Player drawing surface. You can place controls or containers directly under the `<mx:Application>` tag or in other containers. For more information on containers, see [Chapter 4, “Introducing Containers,” on page 161](#).

Most controls have the following characteristics:

- MXML API for declaring the control and the values of its properties and events
- ActionScript API for calling the control’s methods and setting its properties at runtime
- Customizable appearance using styles, skins, and fonts

The following figure shows several controls used in a Form container:



The MXML and ActionScript APIs let you create and configure a control. The following MXML code example creates a TextInput control in a Form container:

```
<mx:Form width="300" height="100" >
  ...
  <mx:FormItem label="Card Name">
    <mx:TextInput id="cardName"/>
  </mx:FormItem>
  ...
</mx:Form>
```

While you commonly use MXML as the language for building Flex applications, you can also use ActionScript to configure controls. For example, the following code populates a DataGrid control by providing an Array of items as the value of the DataGrid control’s `dataProvider` property:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
```



```

<mx:Script>
  <![CDATA[

    function myGrid_initialize(event)
    {
      myGrid.dataProvider =
      [
        { Artist:'Pavement', Album:'Slanted and Enchanted', Price:11.99 },
        { Artist:'Other', Album:'Other', Price:5.99 }
      ];
    }
  ]]>
</mx:Script>
<mx:DataGrid id="myGrid" width="350" height="150"
  initialize="myGrid_initialize();" color="#7B0974"/>
</mx:Application>

```

Using data provider controls

Several Flex components, such as the DataGrid, Tree, and ComboBox controls, take input data from a data provider. A *data provider* is a collection of objects, similar to an array. For example, a Tree control reads data from the data provider to define the structure of the tree and any associated data assigned to each tree node.

The data provider creates a level of abstraction between Flex components and the data that you use to populate them. You can populate multiple components from the same data provider, switch data providers for a component at runtime, and modify the data provider so that changes are reflected by all components using the data provider.

You can think of the data provider as the model, and the Flex components as the view onto the model. By separating the model from the view, you can change one without changing the other.

This chapter describes the basic controls. For information on data provider controls, see [Chapter 3, “Using Data Provider Controls,” on page 91](#).

Flex controls

The following table lists all the controls available with Flex, both basic and data provider:

| Control | Description | For more information |
|-------------|--|--|
| Button | Displays a variable-size button that can include a label, an icon image, or both. | “Button control” on page 37 |
| CheckBox | Shows whether a particular Boolean value is <code>true</code> (checked) or <code>false</code> (unchecked). | “CheckBox control” on page 40 |
| ComboBox | Displays a drop-down list attached to a text field that contains a set of values. | “ComboBox control” on page 141 |
| DataGrid | Displays data in a tabular format. | “DataGrid control” on page 127 |
| DateChooser | Displays a full month of days to let you select a date. | “DateChooser control” on page 42 |

| Control | Description | For more information |
|---------------------------------------|---|--|
| DateField | Displays the date with a calendar icon on its right side. When a user clicks anywhere inside the control, a DateChooser control pops up and displays a month of dates. | “DateField control” on page 49 |
| HorizontalList | The HorizontalList control displays a horizontal list of items. | “HorizontalList control” on page 116 |
| HRule/VRule | Displays a single horizontal rule (HRule) or vertical rule (VRule). | “HRule and VRule controls” on page 53 |
| HSlider/VSlider | Lets users select a value by moving a slider thumb between the end points of the slider track. | “HSlider and VSlider controls” on page 56 |
| Image | Imports a JPEG, PNG, GIF, or SVG image or SWF file. | “Properties of the <mx:Image> tag” on page 282 |
| Label | Displays a noneditable single-line field label. | “Label control” on page 63 |
| Link | Displays a simple hypertext link. | “Link control” on page 68 |
| List | Displays a scrollable array of choices. | “List control” on page 105 |
| Loader | Displays the contents of a specified SWF or JPEG file. | “Loader control” on page 70 |
| Menu | Displays a pop-up menu of individually selectable choices, much like the File or Edit menu of most software applications. | “Menu control” on page 148 |
| MenuBar | Displays a horizontal menu bar that contains one or more submenus of Menu controls. | “MenuBar control” on page 153 |
| NumericStepper | Displays a dual button control you can use to increase or decrease the value of the underlying variable. | “NumericStepper control” on page 72 |
| ProgressBar | Provides visual feedback of how much time remains in the current operation. | “ProgressBar control” on page 74 |
| RadioButton | Displays a set of buttons of which exactly one is selected at any time. | “RadioButton control” on page 77 |
| RadioButton Group | A group of RadioButton controls with a single click handler. | “RadioButtonGroup control syntax” on page 80 |
| ScrollBar (HScrollBar and VScrollBar) | Displays horizontal and vertical scroll bars. Combines a NumericStepper control and a slider control to implement the scrolling functionality in the attached data element. | “ScrollBar control” on page 81 |
| Text | Displays a noneditable multiline text field. | “Text control” on page 82 |

| Control | Description | For more information |
|-----------|--|--|
| TextArea | Displays an editable text field for user input that can accept more than a single line of input. | “TextArea control” on page 85 |
| TextInput | Displays an editable text field for a single line of user input. Can contain alphanumeric data, but input will be interpreted as a String data type. | “TextInput control” on page 88 |
| TileList | The TileList control displays a tiled list of items. The items are tiled in vertical columns or horizontal rows. | “TileList control” on page 121 |
| Tree | Displays hierarchical data arranged as an expandable tree. | “Tree control” on page 135 |

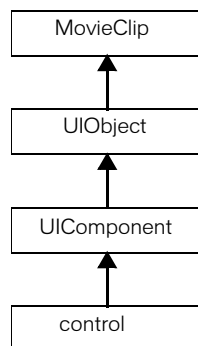
Working with controls

Flex controls share a common class hierarchy. Therefore, you use a similar procedure to configure all controls. This section describes the following topics:

- [“Class hierarchy of controls” on page 35](#)
- [“Sizing controls” on page 36](#)
- [“Positioning controls” on page 37](#)
- [“Changing the appearance of controls” on page 37](#)

Class hierarchy of controls

Flex controls are ActionScript objects derived from the `mx.core.UIObject` and `mx.core.UIComponent` classes, as the following figure shows. Controls inherit the properties, methods, and events of these objects.



The `MovieClip` and `UIObject` classes are the base classes for all Flex components. Subclasses of the `UIObject` class can have shape, draw themselves, and be invisible. The `UIComponent` class is the base class for all Flex components other than the `Label` and `ProgressBar` controls, which are direct subclasses of the `UIObject` class. A class derived from the `UIComponent` class can participate in tabbing, accept low-level events like keyboard and mouse input, and be disabled so that it does not receive mouse and keyboard input.

For more information on the interfaces inherited by controls from the `MovieClip`, `UIObject`, and `UIComponent` classes, see [Chapter 1, “Using Flex Components,” on page 15](#).

Sizing controls

All controls define rules for determining their size in a Flex application. For example, a `Button` control sizes itself to fit its label text, while an `Image` control sizes itself to the size of the imported image. In Flex, the default size of a control is called its *preferred* size, and each control has a preferred height and a preferred width. The preferred size of each standard control is specified in the description of each control.

The preferred size of a control is not necessarily a fixed value. For example, for a `Button` control, the preferred size is a size large enough to fit its text label. At runtime, Flex calculates the preferred size of each control and, by default, does not resize a control from its preferred size. Set the `height` or `width` properties to percentages to allow Flex to resize the control in the corresponding direction. Flex attempts to fit the control to the percentage of its parent container that you specify. If there isn't enough space available, the percentages are scaled, while retaining their relative values.

For instance, you can set the width of a comments box to scale with its parent container:

```
<mx:TextInput id="comments" width="100%" height="20" />
```

For more information on these properties, see [Chapter 4, “Introducing Containers,” on page 161](#).

You can also specify explicit sizes for a control using its `height` and `width` properties. You set the height and width of the `addr2` `TextInput` control in the following example to 20 pixels and 100 pixels, respectively:

```
<mx:TextInput id="addr2" width="100" height="20" />
```

To resize a control at runtime, set its `width` and `height` properties. For example, the click event handler for the following `Button` control sets the `width` property of the `addr2` `TextInput` control to increase its width by 10 pixels:

```
<mx:Button id="button1" label="Slide" height="20"
  click="addr2.width+=10;" />
```

Note: This still works even if the `width` property was originally set as a percentage value. The stored values of the `width` and `height` properties are always in pixels.

Many resizable components have undefined maximum sizes, which means that Flex can make them as large as necessary to fit the requirements of your application. While some components have a defined nonzero minimum size, most have a minimum size of 0. You can use the `maxHeight`, `maxWidth`, `minHeight`, and `minWidth` properties to set explicit size ranges for each component.

Positioning controls

You place controls inside containers. Only a Canvas container allows absolute positioning of its children. All other container types have predefined layout rules that determine the position of their children. For more information about layout management, see [Chapter 4, “Introducing Containers,”](#) on page 161.

To absolutely position a control in a Canvas container, you set its `x` and `y` properties to specific horizontal and vertical pixel coordinates within the container. These coordinates are relative to the upper-left corner of the Canvas container, where the upper-left corner is at coordinates (0,0). Values for `x` and `y` can be positive or negative integers. You can use negative values to place a control outside of the visible area of the container, then use `ActionScript` to move the child to the visible area, possibly as a response to an event.

The following example places the `TextInput` control 150 pixels to the right and 150 pixels down from the upper-left corner of the Canvas container:

```
<mx:TextInput id="addr2" width="100" height="20" x="150" y="150" />
```

To reposition a control within a Canvas container at runtime, you set its `x` and `y` properties. For example, the `click` handler for the following `Button` control moves the `TextInput` control down 10 pixels from its current position:

```
<mx:Button id="button1" label="Slide" height="20" x="0" y="250"
  click="addr2.y = addr2.y+10);" />
```

Changing the appearance of controls

Styles, skins, and fonts let you customize the appearance of controls. They describe aspects of components that you want components to have in common. Typically, you configure things like font family, text alignment, and color. Each control defines a set of styles, skins, and fonts that you can set; some are specific to a particular type of control, while others are more general.

For example, you can set styles for a specific control in the control's MXML tag or using `ActionScript`, or globally for all instances of a specific control using the `<mx:Style>` tag for an entire application.

The current theme for your application defines the styles that you can set on the controls within it. That means some style properties might not always be settable. For more information, see [Chapter 16, “Using Styles and Fonts,”](#) on page 395.

Button control

The `Button` control is a commonly used rectangular button. Button controls look like they can be pressed, and have a text label, an icon, or both on their face. Button controls typically perform an action when clicked, but toggle-style Button controls stay pressed when clicked and act like a radio button. When a user clicks the mouse on a Button control, it broadcasts a `click` event. You can customize the look of a Button control and change its functionality from push to toggle.

The following figure shows a Button control:



The Button control has the following default properties:

| Property | Default |
|----------------|---|
| preferred size | A size large enough to hold the label text, and any icon. |
| minimum size | 0 |
| maximum size | No limit |

Creating a Button control

You define a Button control in MXML using the `<mx:Button>` tag, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block. The following code creates a Button control with the label “Hello world!”:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:Button id='button1' label="Hello world!" width="100"/>
</mx:Application>
```

A Button control’s icon, if specified, and label are centered within the bounds of the Button control. You can position the text label in relation to the icon using the `labelPlacement` property, which accepts the values `right`, `left`, `bottom`, and `top`.

Embedding an icon in a Button control

Flex lets you import graphics into your applications at both compile time and runtime. Button icons must be embedded at compile time rather than referenced at runtime. You can use the `@Embed` syntax in the `icon` property value to embed any JPEG, GIF, SVG, SWF, or PNG file, or you can bind to an image that you defined within a script block by using `[Embed]`. If you must reference your button graphic at runtime, you can use an `<mx:Image>` tag instead of `<mx:Button>`.

Sizing a Button control

By default, Flex stretches the Button control width to fit the size of its label, any icon, plus a 6-pixel margin around the icon. You can override this default width by explicitly setting the `width` property to a specific value or to a percentage of its parent container. If you specify a percentage value, the button resizes between its minimum and maximum widths as the size of its parent container changes.

If you explicitly size a Button control so that it is not large enough to accommodate its label, Flex clips the label. Text that is vertically larger than the Button control is clipped. If you explicitly size a Button control so that it is not large enough to accommodate its icon, icons larger than the Button control extend outside the Button control’s bounding box.

User interaction

When a user clicks the mouse on a Button control, the Button control broadcasts a `click` event, as the following example shows:

```
<?xml version="1.0"?>
<!-- ?xml tag must start in line 1 column 1. -->

<!-- MXML root element tag. -->
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

    <!-- Flex controls exist in a container. Define a vertical Box container for
    the application. -->
    <mx:VBox id="myVbox">
        <!-- input field -->
        <mx:TextInput id="myInput" width="150" text="" />

        <!-- Button control that triggers the copy. -->
        <mx:Button id="myButton" label="Copy Text"
            click="myText.text=myInput.text" />

        <!-- Output text box. -->
        <mx:TextArea id="myText" text="" width="150" height="20" />

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

In this example, clicking the Button control copies the text from the TextInput control to the TextArea control.

If a Button control is enabled, it behaves as follows:

- When the user moves the mouse pointer over the Button control, the Button control displays its roll-over appearance.
- When the user clicks the Button control, focus moves to the control and the Button control displays its pressed appearance. When the mouse button is released, the Button control returns to its roll-over appearance.
- If the user moves the mouse pointer off the Button control while pressing the mouse button, the control's appearance returns to the original state and retains focus.
- If the `toggle` property is set to `true`, the state of the Button control does not change until the mouse button is released over the control.

If a Button control is disabled, it displays its disabled appearance, regardless of user interaction. In the disabled state, all mouse or keyboard interaction is ignored.

Button control syntax

You use the `<mx:Button>` tag to define a Button control in MXML. For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*. The following table describes the optional properties defined by the Button control:

Note: The Button control does not respond to the `valueCommitted` event, even though it is defined in the `UIComponent` class.

| Property | Type | Use | Description |
|-----------------------------|---------|----------|--|
| <code>label</code> | String | Property | Specifies the text label for the control. By default, the label appears centered within the control. |
| <code>labelPlacement</code> | String | Property | Specifies the orientation of the label in relation to a specified icon. Possible values are <code>right</code> (default), <code>left</code> , <code>bottom</code> , and <code>top</code> . |
| <code>icon</code> | File | Property | <p>Specifies the URL to an image file for an icon that appears in the control. Image types include JPEG, GIF, PNG, and SWF. You use the following format with this property:</p> <pre>icon="@Embed('relativeURL')"</pre> <p>The referenced image is packaged within the generated SWF file at compile-time when Flex creates the SWF file for your application.</p> <p>If you specify a SWF file, it cannot contain any ActionScript 2 classes or Macromedia components. If it does, Flex will not embed the SWF file.</p> |
| <code>selected</code> | Boolean | Property | Specifies whether the button is toggled, <code>true</code> , or not, <code>false</code> . The default value is <code>false</code> . |
| <code>toggle</code> | Boolean | Property | Specifies whether a button can be toggled, <code>true</code> , or acts like a pushbutton, <code>false</code> . The default value is <code>false</code> . |
| <code>click</code> | | Event | Specifies a handler for click events. |
| <code>buttonDragOut</code> | | Event | Broadcast when the user clicks a Button control, then drags off of it. |
| <code>buttonDown</code> | | Event | Broadcast when the user clicks the Button control. The <code>click</code> event is triggered when the user releases the mouse. |

CheckBox control

The CheckBox control is a commonly used graphical control that can contain a check mark or be unchecked (empty). You can use CheckBox controls wherever you need to gather a set of `true` or `false` values that aren't mutually exclusive. You can add a text label to a CheckBox control and place it to the left, right, top, or bottom of the check box. When a user clicks a CheckBox control or its associated text, the CheckBox control changes its state from checked to unchecked, or from unchecked to checked.

The following figure shows a CheckBox control:

☒ Employee?

A CheckBox control can have one of two enabled states: checked or unchecked. When checked, the CheckBox control contains a check mark. When unchecked, the CheckBox control is empty. A CheckBox control can also have one of two disabled states, checked or unchecked. By default, a disabled CheckBox control displays a different background and check mark color. The label of a CheckBox control is clipped to fit the boundaries of the control.

The CheckBox control has the following default properties:

| Property | Default |
|----------------|---------------------------------------|
| preferred size | A size large enough to hold the label |
| minimum size | 0 |
| maximum size | No limit |

Creating a CheckBox control

You use the `<mx:CheckBox>` tag to define a CheckBox control in MXML, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:VBox>
    <mx:CheckBox width="100" label="Employee?" />
  </mx:VBox>
</mx:Application>
```

User interaction

When the CheckBox control is enabled and the user clicks it, it receives focus and displays its checked or unchecked appearance, depending on its initial state. The entire bounding box of the CheckBox control is the hit area; if the CheckBox control's font is larger than its icon, there are clickable regions above and below the icon.

If the user moves the mouse pointer outside the bounding area of the CheckBox control or its label while pressing the mouse button, the appearance of the CheckBox control returns to its original state and it retains focus. The state of the CheckBox control does not change until the user releases the mouse button over the component.

Users cannot interact with a CheckBox control when it is disabled.

CheckBox control syntax

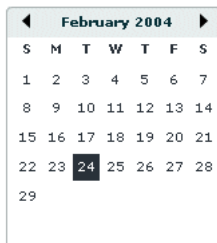
You use the `<mx:CheckBox>` tag to define a CheckBox control. For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*. The following table describes the optional properties defined by the CheckBox control:

| Property | Type | Use | Description |
|----------------|---------|----------|---|
| label | String | Property | Specifies the text label for the control. |
| labelPlacement | String | Property | Specifies the orientation of the label. Possible values are <code>right</code> (default), <code>left</code> , <code>bottom</code> , and <code>top</code> . |
| selected | Boolean | Property | Contains <code>true</code> if the CheckBox control is displaying a check, and <code>false</code> if not. The default value is <code>false</code> . You can read the value of this property to determine whether a user selected the control, or set it to configure the display of the control. |
| click | | Event | Specifies a handler for click events. |
| buttonDragOut | | Event | Broadcast when the user clicks on a check box then drags off of the CheckBox control. |
| buttonDown | | Event | Broadcast when the user clicks the check box. |

DateChooser control

The DateChooser control displays the name of a month, the year, and a grid of the days of the month, with columns labeled for the days of the week. The user can select a single date from the grid. The control contains forward and back arrow buttons to let you change the month and year. You can disable the selection of certain dates, and limit the display to a range of dates.

The following figure shows a DateChooser control:



Changing the displayed month does not change the selected date. Therefore, the currently selected date might not always be visible.

Using the Date class

The Flex DateChooser control uses the ActionScript Date class to represent a date. For more information on the Date class, including its properties and methods, see *Flex ActionScript Language Reference*.

You can create and configure a Date object in MXML using the `<mx:Date>` tag. This tag exposes the setter methods of the Date class as MXML properties so that you can initialize a Date object. For example, the following code creates a DateChooser control, and sets the selected date to April 10, 2003 (note that months are indexed starting at 0 for the DateChooser control):

```
<mx:DateChooser id="date1" >
    <mx:selectedDate>
        <mx:Date month="3" date="10" year="2003" />
    </mx:selectedDate>
</mx:DateChooser>
```

You can also set this property in ActionScript, as the following example shows:

```
<mx:Script>
    <![CDATA[

        function initDC()
        {
            date1.selectedDate= new Date (2003, 3, 10);
        }
    ]]>
</mx:Script>

<mx:DateChooser id="date1" initialize="initDC();" />
```

The MXML properties of the DateChooser control correspond to the list of setter functions, as the following table shows:

| Date method | MXML property | Description |
|----------------------|-----------------|--|
| setDate() | date | Sets the day of the month according to local time. |
| setFullYear() | fullYear | Sets the full year according to local time. |
| setHours() | hours | Sets the hour according to local time. |
| setMilliseconds() | milliseconds | Sets the milliseconds according to local time. |
| setMinutes() | minutes | Sets the minutes according to local time. |
| setMonth() | month | Sets the month according to local time. |
| setSeconds() | seconds | Sets the seconds according to local time. |
| setTime() | time | Sets the time in milliseconds. |
| setUTCDate() | UTCDate | Sets the date according to universal time. |
| setUTCFullYear() | UTCFullYear | Sets the year according to universal time. |
| setUTCHours() | UTCHours | Sets the hour according to universal time. |
| setUTCMilliseconds() | UTCMilliseconds | Sets the milliseconds according to universal time. |
| setUTCMinutes() | UTCMinutes | Sets the minutes according to universal time. |
| setUTCMonth() | UTCMonth | Sets the month according to universal time. |
| setUTCSeconds() | UTCSeconds | Sets the seconds according to universal time. |
| setYear() | year | Sets the year according to local time. |

Creating a DateChooser control

You define a DateChooser control in MXML using the `<mx:DateChooser>` tag, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or an ActionScript block.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

<mx:Script>
  <![CDATA[
    function useDate(eventObj)
    {
      //Access the Date object from the event object.
      day.text=eventObj.target.selectedDate.getDay();
      date.text=eventObj.target.selectedDate.getDate();
      month.text=eventObj.target.selectedDate.getMonth();
      year.text=eventObj.target.selectedDate.getFullYear();
      wholeDate.text=eventObj.target.selectedDate.getFullYear() +
        "/" + (eventObj.target.selectedDate.getMonth()+1) +
        "/" + eventObj.target.selectedDate.getDate();
    }
  ]]>
</mx:Script>

<mx:DateChooser id="date1" change="useDate(event)" />

<mx:Form>
  <mx:FormItem label="Day" >
    <mx:TextInput id="day" width="100" />
  </mx:FormItem >
  <mx:FormItem label="Day of month" >
    <mx:TextInput id="date" width="100" />
  </mx:FormItem >
  <mx:FormItem label="Month" >
    <mx:TextInput id="month" width="100" />
  </mx:FormItem >
  <mx:FormItem label="Year" >
    <mx:TextInput id="year" width="100" />
  </mx:FormItem >
  <mx:FormItem label="Date" >
    <mx:TextInput id="wholeDate" width="300" />
  </mx:FormItem >
</mx:Form>
</mx:Application>
```

This example uses the `change` event of the DateChooser control to display the selected date in several different formats.

The code that determines the value of the `wholeDate` field adds 1 to the month number because the DateChooser control uses a zero-based month system, where January is month 0 and December is month 11.

Setting DateChooser control properties in ActionScript

Properties of the DateChooser control take values that are scalars, arrays, and Date objects. While you can set most of these properties in MXML, it can be easier to set others in ActionScript.

For example, the following code example uses an array to set the `disabledDays` property so that Monday through Thursday are disabled, which means that they cannot be selected in the calendar:

```
<mx:DateChooser id="date1" >
  <mx:disabledDays>
    <mx:Array>
      <mx:String>1</mx:String>
      <mx:String>2</mx:String>
      <mx:String>3</mx:String>
      <mx:String>4</mx:String>
    </mx:Array>
  </mx:disabledDays>
</mx:DateChooser>
```

You can also write this example as the following code shows:

```
<mx:DateChooser id="date1" disabledDays="[0,1,2,3, 4]" />
```

You can also use an array to set the labels of the DateChooser columns using the `dayNames` property, as the following example shows:

```
<mx:DateChooser id="date1" change="useDate(event);" >
  <mx:dayNames>
    <mx:Array>
      <mx:String>Sun</mx:String>
      <mx:String>Mon</mx:String>
      <mx:String>Tues</mx:String>
      <mx:String>Weds</mx:String>
      <mx:String>Th</mx:String>
      <mx:String>Fri</mx:String>
      <mx:String>Sat</mx:String>
    </mx:Array>
  </mx:dayNames>
</mx:DateChooser>
```

You might find it more convenient to set some properties of the DateChooser control in ActionScript. For example, you can set the `disabledDays` property in ActionScript, as the following example shows:

```
<mx:Script>
  <![CDATA[

    function initDC()
    {
      date1.disabledDays=[1,2,3,4];
    }
  ]]>
</mx:Script>

<mx:DateChooser id="date1" initialize="initDC();" />
```

The following example sets the `dayNames`, `firstDayOfWeek`, `headerColor`, and `selectableRange` properties using an `initialize` event:

```
<mx:Script>
  <![CDATA[

    function dateChooser_init()
    {
      myDC.dayNames=['Sun', 'Mon', 'Tue', 'Wed', 'Th', 'Fri', 'Sat'];
      myDC.firstDayOfWeek = 3;
      myDC.setStyle("headerColor", 0xff0000);
      myDC.selectableRange =
        {rangeStart: new Date(2003,1,1), rangeEnd: new Date(2003,3,3)};
    }

    function onScroll(myDC)
    {
      myDC.setStyle("fontStyle", "italic");
    }

  ]]>
</mx:Script>

<mx:DateChooser id = "myDC" width="200" height="200"
  initialize="dateChooser_init();" scroll="onScroll(myDC);" />
```

To set the `selectableRange` property, the code creates two `Date` objects that represent the first date and last date of the range. Users can only select dates within the specified range. This example also changes the `fontStyle` of the `DateChooser` control to *italics* after the first time the user scrolls it.

User interaction

The user can use arrow buttons to advance forward or go back months or years. The user can select a date with the mouse by clicking the desired date.

Clicking a forward month arrow advances a month; clicking the back arrow displays the previous month. Clicking forward a month on December, or back on January, moves to the next (or previous) year. Clicking a date selects it. By default, the selected date is indicated by a dark gray box around the date.

DateChooser control syntax

You use the `<mx:DateChooser>` tag to define a DateChooser control. For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*. The following table describes the optional properties defined by the DateChooser control:

| Property | Type | Use | Description |
|-----------------------------|--------------|----------|---|
| <code>dayNames</code> | String Array | Property | Sets the weekday names for the DateChooser control. Changing this property changes the day labels of the DateChooser control. Sunday is the first day (at index 0) and the rest of the weekday names follow in the normal order. The default value is ["S", "M", "T", "W", "T", "F", "S"]. |
| <code>disabledDays</code> | Number Array | Property | Sets the days to be disabled in a week. All the dates in a month, falling under the specified day, are disabled. This property immediately changes the user interface of the DateChooser control. The elements of this Array can have values between 0 (Sunday) to 6 (Saturday). For example, a value of [0,6] disables Sunday and Saturday. |
| <code>disabledRanges</code> | Array | Property | Disables single as well as multiple days. This property accepts an Array of objects as a parameter. Each object in this Array is: <ul style="list-style-type: none">• A Date object, specifying a single day to disable.• An object containing either or both of two properties: <code>rangeStart</code> and <code>rangeEnd</code>, and each value is a Date object. These properties describe the boundaries of the date range. If either is omitted, the range is considered unbounded in that direction. If you specify only <code>rangeStart</code> , all the dates after the specified date are disabled, including the <code>rangeStart</code> date. If you specify only <code>rangeEnd</code> , all the dates before the specified date are disabled, including the <code>rangeEnd</code> date. If only a single day has to be disabled, use a single Date object that specifies the date. This property immediately changes the user interface of the DateChooser control, if the disabled dates fall inside <code>displayedMonth</code> and <code>displayedYear</code> . |
| <code>displayedMonth</code> | Number | Property | Along with <code>displayedYear</code> , specifies the month displayed in the DateChooser control. Month numbers are zero-based, so January is 0 and December is 11. Changing this property immediately changes the appearance of the component. The default value is the month number of today's date. |
| <code>displayedYear</code> | Number | Property | Along with <code>displayedMonth</code> , determines which year is displayed in the DateChooser control. Setting this property immediately changes the appearance of the component. The default value is today's year. |

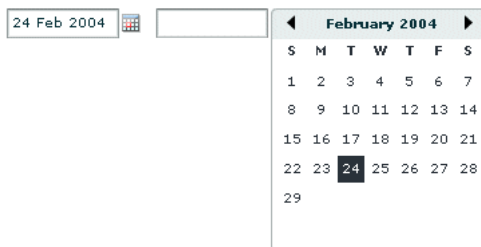
| Property | Type | Use | Description |
|--------------------------------------|--------------|----------|--|
| <code>firstDayOfWeek</code> | Number | Property | Specifies which day of the week (0-6, where 0 is the first element of the <code>dayNames</code> array) is displayed in the first column of the <code>DateChooser</code> control. Changing this property changes the order in which the day columns are displayed. The default value is 0. |
| <code>headerStyleDeclaration</code> | String | Property | Specifies a style sheet definition to configure the date in the header area of the control. If omitted, the header area inherits the text styles of the control. |
| <code>monthNames</code> | String Array | Property | Specifies the names of the months displayed at the top of the <code>DateChooser</code> control. The default value is: ["January", "February", "March", "April", "May", "June", "July", "August", "September", "October", "November", "December"] |
| <code>selectableRange</code> | Array | Property | Sets a range of two dates between which dates are selectable. For example, a date between 04-12-2001 and 04-12-2003 is selectable, but dates out of this range would be disabled. This property accepts an object as a parameter. This object consists of two date objects with variable names <code>rangeStart</code> and <code>rangeEnd</code> . If only <code>rangeStart</code> is defined, all the dates after the defined date are enabled. If only <code>rangeEnd</code> is defined, all the dates before the defined date are enabled. If only a single day has to be enabled in a <code>DateChooser</code> control, a <code>Date</code> object can be passed directly. |
| <code>selectedDate</code> | String | Property | Sets a date as selected in the <code>DateChooser</code> control. Accepts a <code>Date</code> object as a value. |
| <code>showToday</code> | Boolean | Property | If <code>true</code> , specifies that today is highlighted in the <code>DateChooser</code> control. Setting this property immediately changes the appearance of the <code>DateChooser</code> control. The default value is <code>true</code> . |
| <code>todayStyleDeclaration</code> | String | Property | Specifies a style sheet definition to configure the numeric date text of the control. If omitted, the date text field inherits the text styles of the control. |
| <code>weekDayStyleDeclaration</code> | String | Property | Specifies a style sheet definition to configure the weekday names of the control. If omitted, the weekday names inherit the text styles of the control. |
| <code>change</code> | | Event | Specifies an event handler for <code>change</code> events, which are raised whenever a date is selected. |
| <code>scroll</code> | | Event | <ul style="list-style-type: none"> Specifies an event handler for <code>scroll</code> events, which are raised whenever the month changes due to user interaction. |

DateField control

The DateField control is a text field that displays the date with a calendar icon on its right side. When a user clicks anywhere inside the bounding box of the control, a DateChooser control pops up. If no date has been selected, the text field is blank and the month of today's date is displayed in the DateChooser control.

When the DateChooser control is open, users can click the month scroll buttons to scroll through months and years, and select a date. When the user selects a date, the DateChooser control closes and the text field displays the selected date.

The following figure shows a DateField control with the DateChooser control closed, and open. The calendar icon appears on the right side of the DateField text box.



You can use the DateField control anywhere you want a user to select a date. For example, you can use a DateField control in a hotel reservation system, with certain dates selectable and others disabled. You can also use the DateField control in an application that displays current events, such as performances or meetings, when a user selects a date.

Using the Date class

The Flex DateField control uses the ActionScript Date class to represent a date. For more information on using the Date class in Flex, see [“Using the Date class” on page 42](#).

Using the DateChooser control

The DateField control includes the DateChooser control as part of its implementation. For more examples of using the DateChooser control, see [“DateChooser control” on page 42](#).

Creating a DateField control

You define a DateField control in MXML using the `<mx:DateField>` tag, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

This example captures a `change` event when the user selects a date in the DateChooser control, then uses an event handler to populate form fields with the selected date:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

    <mx:Script>
        <![CDATA[
```

```

function useDate(eventObj)
{
    day.text=eventObj.target.selectedDate.getDay();
    date.text=eventObj.target.selectedDate.getDate();
    month.text=eventObj.target.selectedDate.getMonth();
    year.text=eventObj.target.selectedDate.getFullYear();
    wholeDate.text=eventObj.target.selectedDate.getFullYear() + "/" +
        (eventObj.target.selectedDate.getMonth()+1) + "/" +
        eventObj.target.selectedDate.getDate();
}
]]>
</mx:Script>

<mx:DateField id="date1" change="useDate(event)" width="100" />

<mx:Form>
    <mx:FormItem label="Day" >
        <mx:TextInput id="day" width="100" />
    </mx:FormItem >
    <mx:FormItem label="Day of month" >
        <mx:TextInput id="date" width="100" />
    </mx:FormItem >
    <mx:FormItem label="Month" >
        <mx:TextInput id="month" width="100" />
    </mx:FormItem >
    <mx:FormItem label="Year" >
        <mx:TextInput id="year" width="100" />
    </mx:FormItem >
    <mx:FormItem label="Date" >
        <mx:TextInput id="wholeDate" width="300" />
    </mx:FormItem >
</mx:Form>
</mx:Application>

```

Using a date formatter function

By default, the date displayed in the text box is formatted in the form “dd mmm yyyy” where mmm corresponds to the three-letter abbreviation for the month. You can use the `dateFormatter` property of the `DateField` control to specify a function used to format the date displayed in the text box. For example, the following code defines the function `formatDate()` to display the date in the form `yyyy/mm/dd`:

```

<mx:Script>
    <![CDATA[
        function formatDate(date:Date):String
        {
            return date.getUTCDate() + "/" + ( date.getUTCMonth() + 1 ) + "/" +
                date.getUTCFullYear();
        }
    ]]>
</mx:Script>

<mx:DateField id="date1" dateFormatter="formatDate" width="100" />

```

DateField control syntax

You use the `<mx:DateField>` tag to define a DateField control. For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*. The following table describes the optional properties defined by the DateField control:

| Property | Type | Use | Description |
|-----------------------------|--------------|----------|---|
| <code>dateFormatter</code> | Function | Property | Specifies a function used to format the date displayed in the text field of the DateField control. The function receives a Date object as a parameter, and returns a String in the format to be displayed. |
| <code>dayNames</code> | String Array | Property | Sets the weekday names for the DateChooser control. Changing this property changes the day labels of the DateChooser control. Sunday is the first day (at index 0) and the rest of the weekday names follow in the normal order. The default value is ["S", "M", "T", "W", "T", "F", "S"]. |
| <code>disabledDays</code> | Number Array | Property | Sets the days to be disabled in a week. All the dates in a month, falling under the specified day, are disabled. This property immediately changes the user interface of the DateChooser control. The elements of this Array can have values 0 (Sunday) to 6 (Saturday). For example, a value of [0,6] disables Sunday and Saturday. |
| <code>disabledRanges</code> | Array | Property | Disables single as well as multiple days. This property accepts an Array of objects as a parameter. Each object in this Array is: <ul style="list-style-type: none">• A Date Object, specifying a single day to disable.• An object containing either or both of two properties: <code>rangeStart</code> and <code>rangeEnd</code>, each of whose values is a Date object. These properties describe the boundaries of the date range; if either is omitted, the range is considered unbounded in that direction. If you specify only <code>rangeStart</code> , all the dates after the specified date are disabled, including the <code>rangeStart</code> date. If you specify only <code>rangeEnd</code> , all the dates before the specified date are disabled, including the <code>rangeEnd</code> date. If only a single day has to be disabled, use a single Date object specifying the date. This property immediately changes the user interface of the DateChooser control, if the disabled dates fall inside <code>displayedMonth</code> and <code>displayedYear</code> . |
| <code>displayedMonth</code> | Number | Property | Along with <code>displayedYear</code> , specifies the month displayed in the DateChooser control. Months numbers are zero-based, so January is 0 and December is 11. Changing this property immediately changes the appearance of the component. The default value is the month number of today's date. |

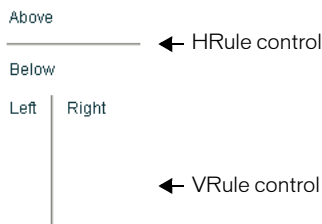
| Property | Type | Use | Description |
|-------------------------------------|--------------------------|----------|--|
| <code>displayedYear</code> | Number | Property | Along with <code>displayedMonth</code> , determines which year is displayed in the <code>DateChooser</code> control. Setting this property immediately changes the appearance of the component. The default value is today's year. |
| <code>firstDayOfWeek</code> | Number | Property | Specifies which day of the week (0-6, 0 being the first element of <code>dayNames</code> array) is displayed in the first column of the <code>DateChooser</code> control. Changing this property changes the order in which the day columns in which they are displayed. The default value is 0. |
| <code>headerStyleDeclaration</code> | String | Property | Specifies a style sheet definition to configure the date in the header area of the control. If omitted, the header area inherit the text styles of the control. |
| <code>monthNames</code> | String Array | Property | Specifies the names of the months displayed at the top of the <code>DateChooser</code> control. The default value is: ["January", "February", "March", "April", "May", "June", "July", "August", "September", "October", "November", "December"] |
| <code>pullDown</code> | <code>DateChooser</code> | Property | Read-only property that contains a reference to the <code>DateChooser</code> control contained by the <code>DateField</code> control. The <code>DateChooser</code> control is instantiated when a user clicks the <code>DateField</code> control. However, if this property is referenced before the user clicks the control, the <code>DateChooser</code> control is instantiated and then hidden. Subsequent clicks result in the use of same instance of the <code>DateChooser</code> control. |
| <code>selectableRange</code> | Array | Property | Sets a range of two dates between which dates are selectable. For example, a date between 04-12-2001 and 04-12-2003 is selectable, but dates out of this range would be disabled. This property accepts an object as a parameter. This object consists of two date objects with variable names <code>rangeStart</code> and <code>rangeEnd</code> . If only <code>rangeStart</code> is defined, all the dates after the defined date are enabled. If only <code>rangeEnd</code> is defined, all the dates before the defined date are enabled. If only a single day has to be enabled in a <code>DateChooser</code> control, a <code>Date</code> object can be passed directly. |
| <code>selectedDate</code> | String | Property | Sets a date as selected in the <code>DateChooser</code> control. Accepts a <code>Date</code> object as the value. |
| <code>showToday</code> | Boolean | Property | If <code>true</code> , specifies that today is highlighted in the <code>DateChooser</code> control. Setting this property immediately changes the appearance of the <code>DateChooser</code> control. The default value is <code>true</code> . |

| Property | Type | Use | Description |
|--|--------|----------|---|
| <code>todayStyle</code> Declaration | String | Property | Specifies a style sheet definition to configure the numeric date text of the control. If omitted, the date text field inherits the text styles of the control. |
| <code>weekDayStyle</code> Declaration | String | Property | Specifies a style sheet definition to configure the weekday names of the control. If omitted, the weekday names inherit the text styles of the control. |
| <code>change</code> | | Event | Specifies an event handler for <code>change</code> events, which are raised whenever a date is selected. Flex also broadcasts a <code>change</code> event if the user closes the control by pressing the Enter key and the selected date has changed, or when the user clicks the mouse pointer outside the control. |
| <code>close</code> | | Event | Specifies an event handler for <code>close</code> events, which are raised whenever a date is selected or the user clicks outside of the pulldown. |
| <code>open</code> | | Event | Specifies an event handler for <code>open</code> events, which are raised whenever a user selects the field to open the pulldown. |
| <code>scroll</code> | | Event | <ul style="list-style-type: none"> Specifies an event handler for <code>scroll</code> events, which are raised whenever the month changes due to user interaction. |

HRule and VRule controls

The HRule (Horizontal Rule) control creates a single horizontal line and the VRule (Vertical Rule) control creates a single vertical line. You typically use these controls to create dividing lines within a container.

The following figure shows an HRule and a VRule control:



The HRule and VRule controls have the following default properties:

| Property | Default |
|--------------------------|--|
| preferred size | Horizontal Rule The default width is 100 pixels, and the default height is 2 pixels. By default, the HRule control is not resizable; set <code>width</code> and <code>height</code> to percentage values to enable resizing. Vertical Rule The default height is 100 pixels, and the default width is 2 pixels. By default, the VRule control is not resizable; set <code>width</code> and <code>height</code> to percentage values to enable resizing. |
| <code>strokeWidth</code> | 2 pixels |
| <code>color</code> | #808080 |
| <code>shadowColor</code> | #D4D0C8 |

Creating HRule and VRule controls

You define an HRule control and VRule control in MXML using the `<mx:HRule>` and `<mx:VRule>` tags, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or an ActionScript block.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

    <mx:VBox>
        <mx:Label text="Above" />
        <mx:HRule />
        <mx:Label text="Below" />
    </mx:VBox>

    <mx:HBox>
        <mx:Label text="Left" />
        <mx:VRule />
        <mx:Label text="Right" />
    </mx:HBox>
</mx:Application>
```

This example creates the output shown in the preceding figure.

You can also use properties of the HRule and VRule controls to specify line width, color, and shadow color, as the following example shows:

```
<mx:VBox>
    <mx:Label text="Above" />
    <mx:HRule shadowColor="#FF0000" />
    <mx:Label text="Below" />
</mx:VBox>

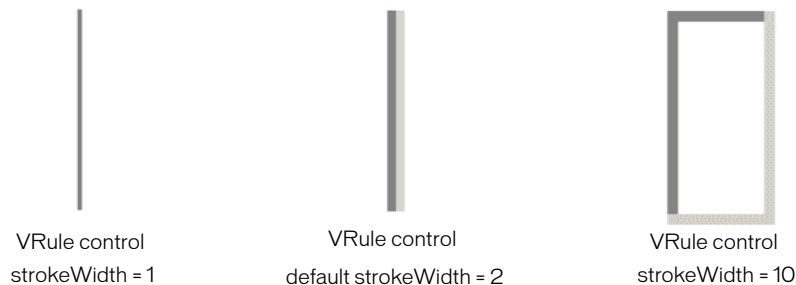
<mx:HBox>
    <mx:Label text="Left" />
    <mx:VRule strokeWidth="10" color="#00FF00" />
    <mx:Label text="Right" />
</mx:HBox>
```

Sizing HRule and VRule controls

For the HRule and VRule controls, the `strokeWidth` property determines how Flex draws the line, as follows:

- If you set the `strokeWidth` property to 1, Flex draws a 1-pixel-wide line.
- If you set the `strokeWidth` property to 2, Flex draws the rule as two adjacent 1-pixel-wide lines, either horizontal for an HRule control or vertical for a VRule control. This is the default value.
- If you set the `strokeWidth` property to a value greater than 2, Flex draws the rule as a hollow rectangle with 1-pixel-wide edges.

The following figure shows all three options:



If you set the `height` property of an HRule control to a value greater than the `strokeWidth` property, Flex draws the rule within a rectangle of the specified height, and centers the rule vertically within the rectangle. The height of the rule is the height specified by the `strokeWidth` property.

If you set the `width` property of a VRule control to a value greater than the `strokeWidth` property, Flex draws the rule within a rectangle of the specified width, and centers the rule horizontally within the rectangle. The width of the rule is the width specified by the `strokeWidth` property.

If you set the `height` property of an HRule control or the `width` property of a VRule control to a value smaller than the `strokeWidth` property, the rule is drawn as if it had a `strokeWidth` property equal to the height or width property.

Note: If the `height` and `width` properties are specified as percentage values, the actual pixel values are calculated before the height and width are compared to the `strokeWidth` property.

The `color` and `shadowColor` properties determine the colors of the HRule and VRule controls. The `color` property specifies the color of the line as follows:

- If you set the `strokeWidth` property to 1, specifies the color of the entire line.
- If you set the `strokeWidth` property to 2, specifies the color of the top line for an HRule control, or the left line for a VRule control.
- If you set the `strokeWidth` property to a value greater than 2, specifies the color of the top and left edges of the rectangle.

The `shadowColor` property specifies the shadow color of the line as follows:

- If you set the `strokeWidth` property to 1, does nothing.
- If you set the `strokeWidth` property to 2, specifies the color of the bottom line for an `HRule` control, or the right line for a `VRule` control.
- If you set the `strokeWidth` property to a value greater than 2, specifies the color of the bottom and right edges of the rectangle.

Setting style properties

The `strokeWidth`, `color`, and `shadowColor` properties are style properties. Therefore, you can set them in MXML as part of the tag definition, set them using the `<mx:Style>` tag in MXML, or set them using the `setStyle()` method in `ActionScript`.

The following example uses the `<mx:Style>` tag to set the default value of the `color` property of all `HRule` controls to `#123456`, and the default value of the `shadowColor` property to `#654321`:

```
<mx:Style>
    .thickRule { strokeWidth: 5 }
    HRule { color: #123456; shadowColor: #654321 }
</mx:Style>
```

This example also defines a class selector, called `thickRule`, with a `strokeWidth` of 5 that you can use with any instance of an `HRule` control or `VRule` control, as the following example shows:

```
<mx:HRule styleName="thickRule" />
```

HRule and VRule control syntax

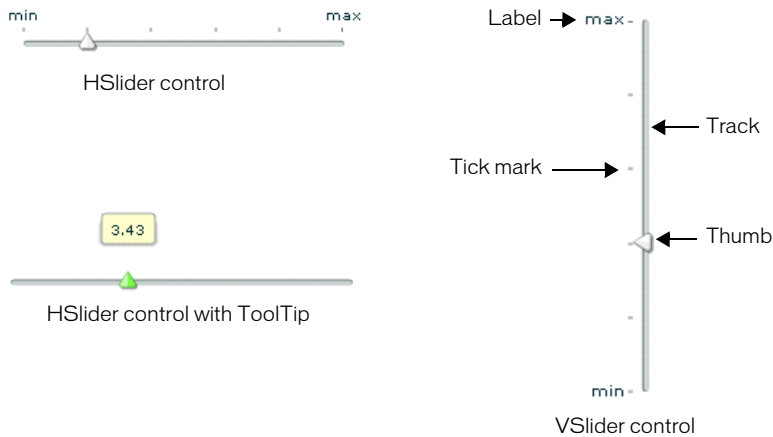
You use the `<mx:HRule>` and `<mx:VRule>` tags to define horizontal and vertical rules. For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*. The following table describes the optional properties defined by these controls:

HSlider and VSlider controls

The slider controls let users select a value by moving a slider thumb between the end points of the slider track. The current value of the slider is determined by the relative location of the thumb between the end points of the slider, corresponding to the slider's minimum and maximum values.

By default, the minimum value of a slider is 0 and the maximum value is 10. The current value of the slider can be any value in a continuous range between the minimum and maximum values, or it can be one of a set of discrete values, depending on how you configure the control.

Flex provides two sliders: the HSlider (Horizontal Slider) control which creates a horizontal slider and the VSlider (Vertical Slider) control creates a vertical slider. The following figure shows an example of the HSlider and VSlider controls:



This figure includes the ToolTip, slider thumb, track, tick marks, and labels. You can optionally show or hide ToolTips, tick marks, and labels.

The HSlider and VSlider controls have the following default properties:

| Property | Default |
|----------------|--|
| preferred size | Horizontal slider 250 pixels wide, and tall enough to hold the slider and any associated labels. Vertical slider 250 pixels tall, and wide enough to hold the slider and any associated labels. |
| minimum size | None |
| maximum size | None |

Creating a Slider control

You define an HSlider control in MXML using the `<mx:HSlider>` tag and a VSlider control using the `<mx:VSlider>` tag. You must specify an `id` value if you intend to refer to a component elsewhere, either in another tag or in an ActionScript block.

The following example creates three HSlider controls:

- The first slider has a maximum value of 100, and lets the user move the slider thumb to select a value in the continuous range from between 0 and 100.
- The second slider uses the `snapInterval` property to define the discrete values between the minimum and maximum that the user can select. In this example, the `snapInterval` is 2, which means that the user can select the values 0, 2, 4, 6, and so on.
- The third slider uses the `tickInterval` property to add tick marks and set the interval between the tick marks to 25, so that Flex displays a tick mark along the slider corresponding to the values 0, 25, 50, 75, and 100. Flex displays tick marks whenever you set the `tickInterval` property to a nonzero value.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:Panel>
    <mx:HSlider maximum="100" />
    <mx:HSlider snapInterval="2" maximum="100" />
    <mx:HSlider snapInterval="2" maximum="100" tickInterval="25"/>
  </mx:Panel>
</mx:Application>
```

The following example creates the VSlider control shown in the previous figure, which includes tick marks and two labels:

```
<mx:VSlider tickInterval="2" labels="['min', 'max']"/>
```

You can bind the `value` property of a slider to another control to display the current value of the slider. The following example binds the `value` property to a Text control:

```
<!-- Bind a slider to a text box. -->
<mx:HSlider id="mySlider" maximum="100" />
<mx:Text text="{mySlider.value}" />
```

Using slider events

The slider controls let the user select a value by moving the slider thumb between the minimum and maximum values of the slider. You use an event with the slider to recognize when the user has moved the thumb, and to determine the current value associated with the slider.

The slider controls can broadcast the events described in the following table:

| Event | Description |
|---------------|---|
| change | Broadcast when the user moves the thumb. If the <code>liveDragging</code> property is true, the event is broadcast continuously as the user moves the thumb. If <code>liveDragging</code> is false, the event is broadcast when the user releases the slider thumb. |
| thumbDragged | Broadcast when the user moves a thumb. |
| thumbPressed | Broadcast when the user selects a thumb using the mouse pointer. |
| thumbReleased | Broadcast when the user releases the mouse pointer after a <code>thumbPressed</code> event occurs. |

The following example uses a `change` event to show the current value of the slider in a TextArea control when the user releases the slider thumb:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

  <mx:Script>
    <![CDATA[
      function sliderChange(event:Object)
      {
        thumb.text=event.target.value;
      }
    ]]>
  </mx:Script>
```

```

<mx:Panel>
  <mx:HSlider change="sliderChange(event)" />
  <mx:TextArea id="thumb" />
</mx:Panel>
</mx:Application>

```

By default, the `liveDragging` property of the slider control is set to `false`, which means that the control broadcasts the change event when the user releases the slider thumb. If you set `liveDragging` to `true`, the control broadcasts the change event continuously as the user moves the thumb, as the following example shows:

```

<?xml version="1.0" encoding="iso-8859-1"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

  <mx:Script>
    <![CDATA[
      function sliderChangeLive(event:Object)
      {
        thumbLive.text=event.target.value;
      }
    ]]>
  </mx:Script>

  <mx:Panel>
    <mx:HSlider liveDragging="true" change="sliderChangeLive(event)" />
    <mx:TextArea id="thumbLive" />
  </mx:Panel>
</mx:Application>

```

Using multiple thumbs

You can configure a slider control to have one thumb, or two thumbs. If you configure the slider to use a single thumb, you can move the thumb anywhere between the end points of the slider. If you configure it to have two thumbs, you cannot drag one thumb across the other thumb.

When you configure a slider control to have two thumbs, you use the `values` property of the control to access the current value of each thumb. The `values` property is a two-element array that contains the current value of the thumbs, as the following example shows:

```

<?xml version="1.0" encoding="iso-8859-1"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:Script>
    <![CDATA[
      function sliderChangeTwo(event:Object)
      {
        thumbTwoA.text=event.target.values[0];
        thumbTwoB.text=event.target.values[1];
        thumbIndex.text=event.thumbIndex;
      }
    ]]>
  </mx:Script>

  <mx:Panel>
    <mx:HSlider thumbCount="2" change="sliderChangeTwo(event)" />
    <mx:TextArea id="thumbTwoA" />

```

```

        <mx:TextArea id="thumbTwoB" />
        <mx:TextArea id="thumbIndex" />
    </mx:Panel>
</mx:Application>

```

This example also uses the `thumbIndex` property of the event object. This property has a value of 0 if the user modified the position of the first thumb, and a value of 1 if the user modified the position of the second thumb.

Using ToolTips

By default, when you select a slider thumb, the slider controls display a ToolTip showing the current value of the slider. As you move the selected thumb, the ToolTip shows the new slider value. You can disable ToolTips by setting the `showToolTip` property to `false`.

You can use the `toolTipFormatFunction` property to specify a callback function to format the ToolTip text. This function takes a single String argument containing the ToolTip text, and returns a String containing the new ToolTip text, as the following example shows:

```

<mx:Script>
    <![CDATA[
        function myToolTipFunc(val:String):String
        {
            return "Current value: " + val;
        }
    ]]>
</mx:Script>

<mx:Panel>
    <!-- Slider with a ToolTip function. -->
    <mx:HSlider toolTipFormatFunction="myToolTipFunc" />
</mx:Panel>

```

In this example, the ToolTip function prepends the ToolTip text with the string "Current value: ". You can modify this example to insert a dollar sign (\$) prefix on the ToolTip for a slider that controls the price of an item.

Keyboard navigation

The HSlider and VSlider controls have the following keyboard navigation features when the slider control has focus:

| Key | Description |
|-------------|---|
| Left Arrow | Decrement the value of an HSlider control by 1 snap interval or, if you do not specify a snap interval, by 1 pixel. |
| Right Arrow | Increment the value of a HSlider control by 1 snap interval or, if you do not specify a snap interval, by 1 pixel. |
| Home | Move the thumb of an HSlider control to its minimum value. |
| End | Move the thumb of an HSlider control to its maximum value. |
| Up Arrow | Increment the value of an VSlider control by 1 snap interval or, if you do not specify a snap interval, by 1 pixel. |

| Key | Description |
|------------|--|
| Down Arrow | Decrement the value of a VSlider control by 1 snap interval or, if you do not specify a snap interval, by 1 pixel. |
| Page Down | Move the thumb of a VSlider control to its minimum value. |
| Page Up | Move the thumb of a VSlider control to its maximum value. |

HSlider and VSlider control syntax

You use the `<mx:HSlider>` tag or `<mx:VSlider>` tag to define a slider control. For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*. The following table describes the optional properties defined by these controls:

| Property | Type | Use | Description |
|------------------------------------|---------|----------|--|
| <code>allowTrackClick</code> | Boolean | Property | Specifies whether clicking on the slider track moves the thumb. The default value is <code>true</code> . |
| <code>labels</code> | Array | Property | Specifies an Array of strings used for the labels. Flex positions labels evenly between the beginning of the track and the end of the track. For example, if the Array contains three labels, Flex positions the first label at the beginning of the track, the second in the middle of the track, and the third at the end of the track. If the Array contains only one label, Flex positions the label at the beginning of the track. The default value is <code>undefined</code> . |
| <code>labelStyleDeclaration</code> | String | Property | Specifies a style sheet definition to configure the slider labels. The default value is <code>""</code> . |
| <code>liveDragging</code> | Boolean | Property | Specifies whether live dragging is enabled for the slider. If <code>false</code> , Flex sets the <code>value</code> and <code>values</code> properties and broadcasts the <code>change</code> event when the user releases the slider thumb. If <code>true</code> , Flex sets the <code>value</code> and <code>values</code> properties and broadcasts the <code>change</code> event continuously as the user moves the thumb. The default value is <code>false</code> . |
| <code>maximum</code> | Number | Property | Specifies the maximum allowed value of the slider. The default value is <code>10</code> . |
| <code>minimum</code> | Number | Property | Specifies the minimum allowed value of the slider. The default value is <code>0</code> . |
| <code>showToolTip</code> | Boolean | Property | Specifies to display ToolTips, <code>true</code> , or not, <code>false</code> . If <code>true</code> , during user interaction, Flex shows a ToolTip with the current value. The default value is <code>true</code> . |

| Property | Type | Use | Description |
|--------------------------------------|----------|----------|--|
| <code>sliderThumbClass</code> | | Property | Specifies a reference to the class to use for each thumb. The default value is <code>mx.controls.sliderclasses.SliderThumb</code> . |
| <code>sliderTooltipClass</code> | | Property | Contains a reference to the class to use for the ToolTip. The default value is <code>mx.controls.sliderclasses.SliderTooltip</code> . |
| <code>snapInterval</code> | Number | Property | Specifies the increment value of the slider thumb as the user moves the thumb. For example, if the <code>snapInterval</code> is 2, the minimum value is 0, and the maximum value is 10, the thumb snaps to the values 0, 2, 4, 6, 8, and 10 as the user moves the thumb. The default value is 0, which means that the slider moves continuously between the minimum and maximum values. |
| <code>thumbCount</code> | Number | Property | Specifies the number of thumbs allowed on the slider. The possible values are 1 or 2. If you set <code>thumbCount</code> to 1, the <code>value</code> property contains the current value of the slider thumb. If you set <code>thumbCount</code> to 2, the <code>values</code> property contains a two-element array of values that contain the value for each thumb. The default value is 1. |
| <code>tickInterval</code> | Number | Property | Specifies the spacing of the tick marks relative to the maximum value of the control. Flex displays tick marks whenever you set the <code>tickInterval</code> property to a nonzero value. For example, if <code>tickInterval</code> is 1, and <code>maximum</code> is 10, Flex places a tick at each 1/10th of the interval along the slider. The default value is 0, which hides the tick marks. |
| <code>tooltipFormatFunction</code> | Function | Property | Specifies a callback function to format the text for the ToolTip. The function takes a single <code>String</code> argument and returns the formatted ToolTip text, as a <code>String</code> . The setting of the <code>tooltipPrecision</code> property is applied to the ToolTip text before it is passed to the function. The signature of the function is: <code>function funcName(val:String):String</code> The default value is undefined. |
| <code>tooltipStyleDeclaration</code> | String | Property | Specifies a style sheet definition to configure the ToolTip. The default value is "". |
| <code>value</code> | Number | Property | Contains the current value of the slider, between the minimum and maximum values. The default value is 0. |

| Property | Type | Use | Description |
|---------------|-------|----------|--|
| values | Array | Property | When the <code>thumbCount</code> is 2, contains a two-element Array of values corresponding to the position of the two thumbs. |
| change | | Event | Broadcast when the slider changes value due to mouse or keyboard interaction. <ul style="list-style-type: none"> If the <code>liveDragging</code> property is <code>true</code>, the event is broadcast continuously as the user moves the thumb. If <code>liveDragging</code> is <code>false</code>, the event is broadcast when the user releases the slider thumb. |
| thumbDragged | | Event | <ul style="list-style-type: none"> Broadcast when the slider is being moved. |
| thumbPressed | | Event | <ul style="list-style-type: none"> Broadcast when the user clicks and holds the mouse on a slider thumb. |
| thumbReleased | | Event | <ul style="list-style-type: none"> Broadcast when the user releases the mouse pointer after a <code>thumbPressed</code> event occurs. |

Label control

The Label control is a noneditable single-line text label. You can specify to format a label as HTML text. You can also control the alignment and sizing of a label. To create a multiline, noneditable text field, use the Text control. For more information, see [“Text control” on page 82](#).

The following figure shows a Label control:

Label1

Label controls do not have borders and cannot take focus. Unlike other controls, Label controls directly extend the `UIObject` class, not the `UIComponent` class.

The Label control has the following default properties:

| Property | Default |
|----------------|--|
| preferred size | Width and height large enough for the text |
| minimum size | 0 |
| maximum size | Undefined |

Creating a Label control

You define a Label control in MXML using the `<mx:Label>` tag, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or an `ActionScript` block.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:Label text="Label1" />
</mx:Application>
```

You use the `text` property to specify a string of raw text, and `htmlText` to specify an HTML-formatted string.

Using the text property

You can use the `text` property to specify the text string that appears in the Label control. The control collapses any white-space characters, such as tab and newline characters. Any HTML tags in the text string are ignored, and appear as entered in the string.

For the special characters left angle bracket (<), right angle bracket (>), and ampersand (&), insert the HTML equivalents of `<`, `>`, and `&`. The following example sets the text string using the `text` property:

```
<mx:Label text="This string contains a less than, &lt;; greater than, &gt;; and  
amp, &amp;." />
```

If you wrap the text string in the `CDATA` tag, you can specify the literal characters for the left angle bracket (<), right angle bracket (>), and ampersand (&) in the `text` property using a child tag, as the following example shows:

```
<mx:Label>  
  <mx:text><![CDATA[This string contains a less than, <, greater than, >,  
    and amp, &. ]]>  
  </mx:text>  
</mx:Label>
```

The following example uses an initialization function to set the `text` property to a string that contains these characters:

```
<mx:Script>  
  <![CDATA[  
  
    function initText() {  
      myLabel.text="This string contains a less than, <, greater than, >,  
        and amp, &."  
    }  
  
  ]]>  
</mx:Script>  
  
<mx:Label id="myLabel" initialize="initText()" />
```

Since the code of the `<mx:Script>` tag is contained in a `CDATA` tag, you do not need an additional tag for the string.

Using the htmlText property

You use the `htmlText` property to specify an HTML-formatted text string. If your text string contains HTML tags, you must wrap it in a `CDATA` tag. The control collapses any white-space characters, such as tab and newline characters.

When you specify the text string for the Label control in MXML, you cannot escape special characters, such as tab and newline characters. For example, if you include the characters `'\t'`, for a tab, in the text string, the characters appear as `'\t'` in the Label control. To insert a tab character, use the `htmlText` property and insert the ` ` escape sequence into the text string.

For the special characters left angle bracket (<), right angle bracket (>), and ampersand (&), insert the HTML equivalents of `<`, `>`, and `&`.

The following example set the text string to an HTML formatted string using the `htmlText` property:

```
<mx:Label >
  <mx:htmlText><![CDATA[<b>This string contains a less than, &lt;, greater
    than, &gt;, and amp, &amp;.</b> ]]>
  </mx:htmlText>
</mx:Label>
```

If you omit the `CDATA` tag, Flex converts the `<`, `>`, and `&` back into the literal characters left angle bracket (`<`), right angle bracket (`>`), and ampersand (`&`), and attempts to interpret them as HTML.

Using HTML-formatted text

Flash Player supports a subset of standard HTML tags such as `<p>` and `` that you can use to format text in any dynamic or input text field.

You must include attributes of HTML tags in double or single quotation marks. Attribute values without quotation marks can produce unexpected results, such as improper rendering of text. For example, the following HTML snippet does not render properly by Flash Player because the value assigned to the `align` attribute (`left`) is not enclosed in quotation marks:

```
textField.htmlText = "<p align=left>This is left-aligned text</p>";
```

If you enclose attribute values in double quotation marks, you must escape the quotation marks (`\`). For example, either of the following is acceptable:

```
textField.htmlText = "<p align='left'>This uses single quotes</p>";
textField.htmlText = "<p align=\"left\">This uses escaped double quotes</p>";
```

You do not need to escape double quotation marks if you're loading text from an external file; it's only necessary if you're assigning a string of text in ActionScript.

Supported HTML tags

This section lists the built-in HTML tags supported by Flash Player.

Anchor tag (`<a>`)

The `<a>` tag creates a hyperlink and supports the following attributes:

- `href` Specifies the URL of the page to load in the browser. The URL can be absolute or relative to the location of the SWF file that is loading the page.
- `target` Specifies the name of the target window to load the page into.

For example, the following HTML snippet creates the link "Go home".

```
<a href="../../../home.htm" target="_blank">Go home</a>
```

You can also define `a:link`, `a:hover`, and `a:active` styles for anchor tags by using style sheets.

Bold tag ()

The tag renders text as bold. A bold typeface must be available for the font used to display the text.

```
<b>This is bold text.</b>
```

Break tag (
)

The
 tag creates a line break in the text field, the following example shows:

```
One line of text<br>Another line of text<br>
```

Note: While you can use a
 tag in the `htmlText` property for the Text, TextArea, and TextInput controls, it is collapsed to a single space character in a Label control because a Label control can only contain a single line of text.

Font tag ()

The tag specifies a font or list of fonts to display the text. You can also specify formatting to the Label control using Flex CSS styles. If you specify conflicting settings using the tag and CSS styles, Flex uses the settings from the tag.

The font tag supports the following attributes:

color Only hexadecimal color (#FFFFFF) values are supported. For example, the following HTML code creates red text:

```
<font color="#FF0000">This is red text</font>
```

face Specifies the name of the font to use. You can also specify a list of comma-separated font names, in which case Flash Player chooses the first available font. If the specified font is not installed on the playback system, or isn't embedded in the SWF file, Flash Player chooses a substitute font.

```
<font face="Times, Times New Roman">This is either Times or Times New  
Roman.</font>
```

size Specifies the size of the font, in pixels. You can also use relative point sizes (for example, +2 or -4).

```
<font size="24" color="#0000FF">This is green, 24-point text</font>
```

Italic tag (<i>)

The <i> tag displays the tagged text in italics. An italic typeface must be available for the font used.

```
That is very <i>interesting</i>.
```

List item tag ()

The tag places a bullet in front of the text that it encloses.

```
Grocery list:  
<li>Apples</li>  
<li>Oranges</li>  
<li>Lemons</li>
```

Paragraph tag (<p>)

The <p> tag creates a new paragraph. It supports the following attribute:

align Specifies alignment of text within the paragraph; valid values are `left`, `right`, and `center`.

The following example uses the `align` attribute to align text on the right side of a text field:

```
textField.htmlText = "<p align='right'>This text is aligned on the right side  
of the text field</p>";
```

Span tag ()

The tag is available only for use with CSS text styles. It supports the following attribute:

- **class** Specifies a CSS style class defined by a `TextField.StyleSheet` object.

Underline tag (<u>)

The <u> tag underlines the tagged text.

This text is <u>underlined</u>.

Label control syntax

You use the <mx:Label> tag to define a Label control. For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*. The following table describes the optional properties defined by the Label control:

| Property | Type | Use | Description |
|----------|--------|----------|--|
| htmlText | String | Property | Contains HTML formatted text. If your text string contains HTML tags, you must wrap it in a <code>CDATA</code> tag. The control collapses any white-space characters, such as tab and newline characters. For the special characters left angle bracket (<), right angle bracket (>), and ampersand (&), insert the HTML equivalents of <code>&lt;</code> , <code>&gt;</code> , and <code>&amp;</code> . The <code>htmlText</code> property ignores CSS style settings for the control, and instead relies on the HTML tags in the string for formatting. You should not try to mix HTML tags and styles; use HTML tags to format your text. |
| text | String | Property | Specifies the text string that appears in the Label control. The control collapses any white-space characters, such as tab and newline characters. Any HTML tags in the text string are ignored, and appear as entered in the string. For the special characters left angle bracket (<), right angle bracket (>), and ampersand (&), wrap the text string in a <code>CDATA</code> tag. |

Link control

The Link control creates a single-line hypertext link that supports an optional icon. You can use a Link control to open a URL in a web browser.

The following figure shows three Link controls:

Link1 **Link2** Link3

The Link control has the following default properties:

| Property | Default |
|----------------|--|
| preferred size | Width and height large enough for the text |
| minimum size | 0 |
| maximum size | Undefined |

Creating a Link control

You define a Link control in MXML using the `<mx:Link>` tag, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The following code contains a single Link control that opens a URL in a web browser window:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:Link label="MACR" width="100"
    click="getURL('http://quote.yahoo.com/q?s=MACR', 'quote')"
  />
</mx:Application>
```

This example uses the ActionScript `getURL()` function to open the URL. For more information, see *Flex ActionScript Language Reference*.

The following code contains Link controls for navigating in a ViewStack container:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

  <mx:VBox>
    <!-- Put the links in an HBox container across the top. -->
    <mx:HBox>
      <mx:Link label="Link1" click="viewStack.selectedIndex=0" />
      <mx:Link label="Link2" click="viewStack.selectedIndex=1" />
      <mx:Link label="Link3" click="viewStack.selectedIndex=2" />
    </mx:HBox>

    <!-- This ViewStack container has three children. -->
    <mx:ViewStack id="viewStack" >
      <mx:VBox>
        <mx:Label text="One"/>
      </mx:VBox>
      <mx:VBox>
        <mx:Label text="Two"/>
      </mx:VBox>
    </mx:ViewStack>
  </mx:VBox>
</mx:Application>
```

```

        </mx:VBox>
        <mx:VBox>
            <mx:Label text="Three" />
        </mx:VBox>
    </mx:ViewStack>
</mx:VBox>
</mx:Application>

```

A Link control's label is centered within the bounds of the Link control. You can position the text label in relation to the icon using the `labelPlacement` property, which accepts the values `right`, `left`, `bottom`, and `top`.

User interaction

When a user clicks a Link control, the Link control broadcasts a `click` event. If a Link control is enabled, the following happens:

- When the user moves the mouse pointer over the Link control, the Link control displays its rollover appearance.
- When the user clicks the Link control, the input focus moves to the control and the Link control displays its pressed appearance. When the mouse button is released, the Link control returns to its rollover appearance.
- If the user moves the mouse pointer off the Link control while pressing the mouse button, the control's appearance returns to its original state and retains input focus.
- If the `toggle` property is set to `true`, the state of the Link control does not change until the mouse button is released over the control.

If a Link control is disabled, it displays its disabled appearance, regardless of user interaction. In the disabled state, all mouse or keyboard interaction is ignored.

Link control syntax

You use the `<mx:Link>` tag to define a Link control. For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*. The following table describes the optional properties defined by the Link control:

| Property | Type | Use | Description |
|--------------------|--------|----------|--|
| <code>icon</code> | File | Property | Specifies the URL of a JPEG, GIF, SVG, or PNG image or SWF file used as the icon. You use the following format with this property: <code>icon="@Embed('relativeAbsoluteURL')"</code> The referenced image is packaged within the generated SWF file at compile time when Flex creates the SWF file for your application. If you specify a SWF file, it cannot contain any ActionScript 2 classes or Macromedia components. If it does, Flex does not embed the SWF file. |
| <code>label</code> | String | Property | Specifies the text label for the control. By default, the label appears centered in the Link control. |

| Property | Type | Use | Description |
|----------------|--------|----------|--|
| labelPlacement | String | Property | Specifies the orientation of the label in relation to a specified icon. Possible values are <code>right</code> (default), <code>left</code> , <code>bottom</code> , and <code>top</code> . |
| click | | Event | Specifies a handler for <code>click</code> events. |

Loader control

The Loader control displays the contents of a specified SWF or JPEG file. It has properties that let you scale its contents. It can also resize itself to fit the size of its contents. By default, content is scaled to fit the size of the Loader control. It can also load content on demand programmatically, and monitor the progress of a load.

A Loader control cannot receive focus. However, content loaded into the Loader control can accept focus and have its own focus interactions.

Although the Loader is essentially a control, it extends the View class. For more information about the View class, see [Chapter 4, “Introducing Containers,” on page 161](#).

The Loader control has the following default properties:

| Property | Default |
|----------------|--|
| preferred size | Width and height large enough for the loaded content |
| minimum size | 0 |
| maximum size | Undefined |

Creating a Loader control

You define a Loader control in MXML using the `<mx:Loader>` tag, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:Loader id="loader1" contentPath="artwork.swf"/>
</mx:Application><mx:HBox id="myhbox">
```

Using the Loader control to load a Flex application

You can use the Loader control to load a Flex application. The following example loads the file `myApp.mxml`:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

  <mx:VBox />
  <mx:Loader source="myApp.mxml.swf" scaleContent="false" />
  </mx:VBox />
</mx:Application>
```

This technique works well with SWF files that add graphics or animations to an application, but are not intended to have a large amount of user interaction. If you want to import SWF files that require a large amount of user interaction, you should build them as custom components.

Sizing a Loader control

You use the Loader control's `scaleContent` property to control the sizing behavior of the Loader control. When the `scaleContent` property is set to `true`, Flex scales the content to fit within the bounds of the control.

Loader control syntax

You use the `<mx:Loader>` tag to define a Loader control. For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*. The following table describes the optional properties defined by the Loader control:

| Property | Type | Use | Description |
|----------------------------|---------|----------|--|
| <code>autoLoad</code> | Boolean | Property | Specifies whether the content loads automatically, <code>true</code> , or if you must call the load method, <code>false</code> . The default value is <code>true</code> . |
| <code>brokenImage</code> | Path | Property | <p>Specifies the image to use if the Loader control cannot load the image specified by <code>contentPath</code>. By default, Flex uses a standard broken image icon.</p> <p>You must use the <code>@Embed</code> tag to specify the image, as the following code shows:</p> <pre><mx:Loader brokenImage="@Embed('myImage.jpg')"/> /></pre> <p>For more information on using <code>@Embed</code>, see Chapter 9, "Importing Images," on page 281.</p> <p>If you specify a SWF file, it cannot contain any ActionScript 2 classes or Macromedia components. If it does, Flex does not embed the SWF file.</p> |
| <code>bytesLoaded</code> | Number | Property | Read-only property that contains the number of bytes loaded. |
| <code>bytesTotal</code> | Number | Property | Read-only property that contains the total number of bytes in the content. |
| <code>content</code> | File | Property | Read-only property that contains the content of the Loader control. |
| <code>contentPath</code> | Path | Property | Specifies the absolute or relative URL of the content to be loaded. A relative URL is relative to the directory that contains the file using the Loader control. |
| <code>percentLoaded</code> | Number | Property | Read-only property that contains the percentage of the content that has been loaded. |
| <code>scaleContent</code> | Boolean | Property | Specifies whether the content scales to fit the Loader control, <code>true</code> , or the Loader control scales to fit the content, <code>false</code> . The default value is <code>true</code> . |

| Property | Type | Use | Description |
|----------|------|-------|--|
| complete | | Event | Specifies a handler for <code>complete</code> events, which are broadcast when loading is completed. |
| progress | | Event | Specifies a handler for an event triggered while content is loading. The event is not guaranteed to be dispatched, which means that the <code>complete</code> event might be received, without any <code>progress</code> events being dispatched. |

NumericStepper control

The NumericStepper control lets the user select a number from an ordered set. The NumericStepper control consists of a single-line input text field and a pair of arrow buttons for stepping through the possible values; the user can also use the Up and Down Arrow keys to cycle through the values.

The following figure shows a NumericStepper control:



If the user clicks the up arrow, the value displayed is increased by one unit of change. If the user holds down the arrow, the value increases or decreases until the user releases the mouse button. When the user clicks the arrow, it is highlighted to provide feedback to the user.

Users can also type a legal value directly into the stepper. Although editable ComboBox controls provide similar functionality, NumericStepper controls are sometimes preferred because they do not require a drop-down list that can obscure important data.

NumericStepper control arrows always appear to the right of the text field.

The NumericStepper control has the following default properties:

| Property | Default |
|----------------|---|
| preferred size | Wide enough to display the maximum number of digits used by the control |
| minimum size | Based on the size of the text |
| maximum size | Undefined |

Creating a NumericStepper control

You define a NumericStepper control in MXML using the `<mx:NumericStepper>` tag, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
    ...
    <mx:NumericStepper id="nstepper1" value="6" stepSize="2" />
    ...
</mx:Application>
```


Sizing a NumericStepper control

The up and down arrow buttons in the NumericStepper control do not change size when the control is resized. If the NumericStepper control is sized greater than the default height, the associated stepper buttons appear pinned to the top and the bottom of the control.

User interaction

If the user clicks the up or down arrow button, the value displayed is increased by one unit of change. If the user presses either of the arrow buttons for more than 200 milliseconds, the value in the input field increases or decreases, based on step size, until the user releases the mouse button or the maximum or minimum value is reached.

Keyboard navigation

The NumericStepper control has the following keyboard navigation features:

| Key | Description |
|-------|---|
| Down | Value decreases by one unit. |
| Up | Value increases by one unit. |
| Left | Moves the insertion point to the left within the NumericStepper control's text field. |
| Right | Moves the insertion point to the right within the text field. |

NumericStepper control syntax

You use the `<mx:NumericStepper>` tag to define a NumericStepper control. For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*. The following table describes the optional properties defined by the NumericStepper control:

| Property | Type | Use | Description |
|----------------------------|--------|----------|---|
| <code>maximum</code> | Number | Property | Specifies the maximum range value. Can contain a number with up to three decimal places. The default value is 10. |
| <code>minimum</code> | Number | Property | Specifies the minimum range value. The default value is 0. |
| <code>nextValue</code> | Number | Property | Read-only property that contains the next value in the control, based on the range and step size. |
| <code>previousValue</code> | Number | Property | Read-only property that contains the previous value in the control, based on the range and step size. |
| <code>stepSize</code> | Number | Property | Specifies the nonzero unit change from the current value. Can contain a number with up to three decimal places. The default value is 1. |

| Property | Type | Use | Description |
|----------|--------|----------|--|
| value | Number | Property | Sets the current value displayed in the text field of the NumericStepper control. The value is not assigned if it does not correspond to the NumericStepper control's <code>maximum</code> and <code>minimum</code> range values and <code>stepSize</code> . Can contain a number with up to three decimal places. The default value is 0. |
| change | | Event | Specifies a handler for <code>change</code> events, which are broadcast when the value of the NumericStepper control changes as a result of user interaction. |

ProgressBar control

The ProgressBar control provides a visual representation of the progress of a task over time. There are two types of ProgressBar controls: determinate and indeterminate. A *determinate* ProgressBar control is a linear representation of the progress of a task over time. You can use this when the user is required to wait for an extended period of time, and the scope of the task is known.

An *indeterminate* ProgressBar control represents time-based processes for which the scope is not yet known. As soon as you can determine the scope, you should use a determinate ProgressBar control.

The following figure shows both types of ProgressBar controls:



Use the ProgressBar control when the user is required to wait for completion of a process over an extended period of time. You can attach the ProgressBar control to any kind of loading content. A label can display the extent of loaded contents when enabled.

The ProgressBar control has the following default properties:

| Property | Default |
|----------------|-----------------------------------|
| preferred size | 150 pixels wide and 4 pixels high |
| minimum size | 0 |
| maximum size | Undefined |

ProgressBar control modes

You use the `mode` property to specify the operating mode of the ProgressBar control. The ProgressBar control supports the following modes of operation:

event Use the `source` property to specify a loading process that emits `progress` and `complete` events. For example, the Loader and Image controls emit these events as part of loading an image. This is the default mode. You typically use a determinate ProgressBar in this mode.

You also use this mode if you want to measure progress on multiple loads; for example, if you reload an image, or use the Loader and Image controls to load multiple images.

polled Use the `source` property to specify a loading process that exposes the `getBytesLoaded()` and `getBytesTotal()` methods. For example, the `Loader` and `Image` controls expose these methods. You typically use a determinate `ProgressBar` in this mode.

manual Set the `maximum`, `minimum`, and `indeterminate` properties along with calls to the `setProgress()` method. You typically use an indeterminate `ProgressBar` in this mode.

Creating a `ProgressBar` control

You use the `<mx:ProgressBar>` tag to define a `ProgressBar` control in MXML, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an `ActionScript` block.

This example uses the default event mode to track the progress of loading an image using the `Loader` control.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

    <mx:Script>
        <![CDATA[
            function initImage()
            {
                image1.load('bigimage.jpg');
            }
        ]]>
    </mx:Script>

    <mx:VBox id="vbox0" width="600" height="600">
        <mx:Canvas>
            <mx:ProgressBar width="200" source="image1"/>
        </mx:Canvas>
        <mx:Button id="myButton" label="Show" click="initImage()"/>
        <mx:Loader height="600" width="600" id="image1"
            autoLoad="false" visible="true" />
    </mx:VBox>
</mx:Application>
```

In this mode, the `Loader` control issues progress events during the load, and a complete event when the load completes.

Since the `<mx:Loader>` tag exposes the `getBytesLoaded()` and `getBytesTotal()` methods, you could also use `polled` mode, as the following example shows:

```
<mx:ProgressBar width="200" source="image1" mode="polled" />
```

In `manual` mode, `mode="manual"`, you use an indeterminate `ProgressBar` control with the `maximum` and `minimum` properties and the `setProgress()` method. The `setProgress()` method has the following method signature:

```
setProgress(Number completed, Number total)
```

where:

completed Specifies the progress made in the task, and must be between the `maximum` and `minimum` values. For example, if you were tracking the number of bytes to load, this would be the number of bytes already loaded.

total Specifies the total task. For example, if you were tracking bytes loaded, this would be the total number of bytes to load. Typically, this is the same value as `maximum`.

To measure progress, you make explicit calls to the `setProgress()` method to update the `ProgressBar` control.

Defining the label of a `ProgressBar` control

By default, the `ProgressBar` displays the label *LOADING xx%* where *xx* is the percent of the image loaded. You use the `label` property to specify a different text string to display.

The `label` property lets you include the following special characters in the label text string:

%1 Corresponds to the current number of bytes loaded.

%2 Corresponds to the total number of bytes.

%3 Corresponds to the percent loaded.

%% Corresponds to the % sign.

For example, to define a label that displays as:

Loading Image 1500 out of 78000 bytes, 2%

Use the following code:

```
<mx:ProgressBar width="300" source="image1" mode="polled"
  label="Loading Image %1 out of %2 bytes, %3%" />
```

ProgressBar control syntax

You use the `<mx:ProgressBar>` tag to define a `ProgressBar` control. For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*. The following table describes the optional properties and methods defined by the `ProgressBar` control:

| Property/Method | Type | Use | Description |
|-------------------------|--------|----------|---|
| <code>conversion</code> | Number | Property | Specifies the number used to convert the incoming current bytes loaded value and the total bytes loaded values. Flex divides the current and total values and floors them before displaying them in the label string. The floor is the closest integer that is less than or equal to the specified value. The default value is 1, which does no conversion. |
| <code>direction</code> | String | Property | Specifies the direction of fill of the progress bar. Valid values are <code>right</code> and <code>left</code> . The default value is <code>right</code> . |

| Property/Method | Type | Use | Description |
|------------------------------|---------|----------|--|
| <code>indeterminate</code> | Boolean | Property | Specifies whether the <code>ProgressBar</code> control has a determinate or indeterminate appearance. Appearance is indeterminate when set to <code>true</code> . The default value is <code>false</code> . |
| <code>label</code> | String | Property | Specifies the text that accompanies the progress bar. You can include the following special characters in the text string: <code>%1</code> = current loaded bytes, <code>%2</code> = total bytes, <code>%3</code> = percent loaded, <code>%%</code> = percent symbol. If a field is unknown, it is replaced by <code>'??'</code> . If <code>undefined</code> , the label is not displayed. |
| <code>labelPlacement</code> | String | Property | Specifies the placement of the label. Valid values are <code>top</code> , <code>bottom</code> (default), <code>left</code> , <code>right</code> , and <code>center</code> . |
| <code>maximum</code> | Number | Property | Specifies the largest progress value for the progress bar. You can only use this property in <code>manual</code> mode. The default value is 0. |
| <code>minimum</code> | Number | Property | Specifies the smallest progress value for the progress bar. The developer sets this property only in <code>manual</code> mode. The default value is 0. |
| <code>mode</code> | String | Property | Specifies the mode as <code>event</code> (default), <code>polled</code> , or <code>manual</code> . For more information, see “ProgressBar control modes” on page 74 . |
| <code>percentComplete</code> | Number | Property | Read-only property that contains the percentage of the process completed. |
| <code>source</code> | | Property | Specifies the instance that will be loading content to the bar. This property is used only in <code>event</code> and <code>polled</code> modes. |
| <code>value</code> | Number | Property | Read-only property that contains the progress that has been made, between the <code>minimum</code> and <code>maximum</code> values. |
| <code>complete</code> | | Event | <ul style="list-style-type: none"> Specifies a handler for <code>complete</code> events, which are broadcast when the load completes. |
| <code>progress</code> | | Event | <ul style="list-style-type: none"> Specifies a handler for <code>progress</code> events, which are broadcast as content loads in <code>event</code> or <code>polled</code> mode. |
| <code>setProgress()</code> | | Method | <p>Sets the state of the bar to reflect the amount of progress made when using <code>manual</code> mode.</p> <p>This method has the following method signature:</p> <pre>setProgress(completed:Number, total:Number)</pre> <p>The argument <code>completed</code> is assigned to the <code>value</code> property and the argument <code>total</code> is assigned to the <code>maximum</code> property. The <code>minimum</code> property is not altered.</p> |

RadioButton control

The `RadioButton` control is a single choice in a set of mutually exclusive choices. A `RadioButton` group is composed of two or more `RadioButton` controls with the same group name. Only one member of the group can be selected at any given time. Selecting an unselected group member deselects the currently selected `RadioButton` control in the group.

The following figure shows a RadioButton group with three RadioButton controls:



The RadioButton control has the following default properties:

| Property | Default |
|----------------|--|
| preferred size | Wide enough to display the text label of the control |
| minimum size | 0 |
| maximum size | Undefined |

Creating a RadioButton control

You define a RadioButton control in MXML using the `<mx:RadioButton>` tag, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:RadioButton groupName="cardtype" id="americanExpress"
    label="American Express" width="200"/>
  <mx:RadioButton groupName="cardtype" id="masterCard"
    label="MasterCard" width="200"/>
  <mx:RadioButton groupName="cardtype" id="visa"
    label="Visa" width="200"/>
</mx:Application>
```

For each RadioButton control in the group, you can optionally define a handler for the button's click event. When a user selects a RadioButton control, Flex calls the handler associated with the button for the click event, as the following code example shows:

```
<mx:RadioButton groupName="cardtype" id="americanExpress"
  label="American Express" width="200" click="handleAmEx();" />
<mx:RadioButton groupName="cardtype" id="masterCard"
  label="MasterCard" width="200" click="handleMC();" />
<mx:RadioButton groupName="cardtype" id="visa"
  label="Visa" width="200" click="handleVisa();" />
```

User interaction

If a RadioButton control is enabled, when the user moves the mouse pointer over an unselected RadioButton control, the button displays its roll-over appearance. When the user clicks an unselected RadioButton control, the input focus moves to the control and the button displays its false pressed appearance. When the mouse button is released, the button displays the true state appearance. The previously selected RadioButton control in the group returns to its false state.

If the user moves the mouse pointer off the RadioButton control while pressing the mouse button, the control's appearance returns to the false state and retains input focus.

If a `RadioButton` control is not enabled, the `RadioButton` control and `RadioButton` group display the disabled appearance, regardless of user interaction. In the disabled state, all mouse or keyboard interaction is ignored.

Creating a group using the `<mx:RadioButtonGroup>` tag

The previous example created a `RadioButton` group using the `groupName` property of each `RadioButton` control. You can also create a `RadioButton` group using the `<mx:RadioButtonGroup>` tag, as the following example shows:

```
<mx:RadioButtonGroup id="cardtype" click="handleCard(event)" />
<mx:RadioButton groupName="cardtype" id="americanExpress" data="AmEx"
    label="American Express" width="200" />
<mx:RadioButton groupName="cardtype" id="masterCard" data="MC"
    label="MasterCard" width="200" />
<mx:RadioButton groupName="cardtype" id="visa" data="Visa"
    label="Visa" width="200" />
```

In this example, you use `id` property of the `<mx:RadioButtonGroup>` tag to define the group name and the single `click` handler for all buttons in the group. The `id` property is required when you use the `<mx:RadioButtonGroup>` tag. The `click` handler for the group can determine which button was selected, as the following example shows:

```
<mx:Script>
    <![CDATA[
        function handleCard(evtObj)
        {
            if (evtObj.target.selectedData == "AmEx") {
                // Process AmEx card.
            } else {
                if (evtObj.target.selectedData == "MC") {
                    // Process MC card.
                } else {
                    // Process Visa.
                }
            }
        }
    ]]>
</mx:Script>
```

In the `click` handler, the `selectedData` property of the `RadioButtonGroup` control in the event object is set to the value of the `data` property of the selected `RadioButton` control. If you omit the `data` property, Flex sets the `selectedData` property to the value of the `label` property.

You can still define a `click` handler for the individual buttons even though you also define one for the group.

RadioButton control syntax

You use the `<mx:RadioButton>` tag to define a `RadioButton` control. For a complete description of the syntax, see the *Flex ActionScript and MXML API Reference*. The following table describes the properties defined by the `RadioButton` control:

| Property | Type | Use | Description |
|-----------------------------|---------|----------|--|
| <code>data</code> | Any | Property | Specifies the data value associated with a radio button instance. |
| <code>groupName</code> | String | Property | Specifies the group name for a radio button group or radio button instance. The name refers to the value of the <code>id</code> property of an <code><mx:RadioButtonGroup></code> tag. |
| <code>label</code> | String | Property | Specifies the text label for the control. By default, the label appears to the right of the <code>RadioButton</code> control. |
| <code>labelPlacement</code> | String | Property | Specifies the orientation of the label, relative to the <code>RadioButton</code> icon. Possible values are <code>right</code> , <code>left</code> , <code>bottom</code> , and <code>top</code> . The default value is <code>right</code> . |
| <code>selected</code> | Boolean | Property | Specifies whether the button is selected, <code>true</code> , or not, <code>false</code> . The default value is <code>false</code> . |
| <code>selectedData</code> | Any | Property | Selects the radio button in a radio button group with the specified data value. |
| <code>toggle</code> | Boolean | Property | Specifies whether a <code>RadioButton</code> control can be toggled, <code>true</code> , or acts like a pushbutton, <code>false</code> . The default value is <code>false</code> . |
| <code>click</code> | | Event | Specifies a handler for <code>click</code> events. |

RadioButtonGroup control syntax

You use the `<mx:RadioButtonGroup>` tag to define a `RadioButtonGroup` control. The `id` property is required when you use the `<mx:RadioButtonGroup>` tag to define the name of the group. For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*. The following table describes the optional properties defined by the `RadioButtonGroup` control:

| Property | Type | Use | Description |
|-----------------------------|---------|----------|--|
| <code>enabled</code> | Boolean | Property | Specifies to enable the control, if <code>true</code> . The default value is <code>true</code> . |
| <code>groupName</code> | String | Property | Specifies the group name for a radio button group. This property is for backward compatibility with Macromedia Flash only. Use the <code>id</code> property of the <code><mx:RadioButtonGroup></code> tag to define the name of the group. |
| <code>labelPlacement</code> | String | Property | Specifies the orientation of the label relative to the <code>RadioButton</code> icon for all controls in the group. You can override this setting for the individual controls. Possible values are <code>right</code> , <code>left</code> , <code>bottom</code> , and <code>top</code> . The default value is <code>right</code> . |
| <code>selectedData</code> | Any | Property | Value of the <code>data</code> property of the selected <code>RadioButton</code> in the group. If the <code>RadioButton</code> does not define the <code>data</code> property, Flex sets <code>selectedData</code> to the value of the <code>label</code> property. |

| Property | Type | Use | Description |
|-----------|--------|----------|--|
| selection | Object | Property | Contains a copy of the currently selected RadioButton control in the group. You can only access this property in ActionScript; it is not settable in MXML. |
| click | | Event | Specifies a handler for click events for the group. You can also set a handler for individual RadioButton controls. |

ScrollBar control

The VScrollBar (vertical ScrollBar) control and HScrollBar (horizontal ScrollBar) control let the user control the portion of data that is displayed when there is too much data to fit in the display area.

Although you can use the VScrollBar control and HScrollBar control as stand-alone controls, they are usually combined with other components as part of a custom component to provide scrolling functionality. For more information, see [Chapter 11, “Building an Application with Multiple MXML Files,” on page 309](#).

ScrollBar controls consists of four parts: two arrow buttons, a track, and a thumb. The position of the thumb and display of the buttons depends on the current state of the ScrollBar control. The ScrollBar control uses four parameters to calculate its display state:

- Minimum range value
- Maximum range value
- Current position; must be within the minimum and maximum range values
- Viewport size; represents the number of items in the range that can be displayed at once and must be equal to or less than the range

Creating a ScrollBar control

You define a ScrollBar control in MXML using the `<mx:VScrollBar>` tag for a vertical ScrollBar or the `<mx:HScrollBar>` tag for a horizontal ScrollBar, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

```
<?xml version="1.0" ?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
    ...
    <mx:VScrollBar id="scroll1" scroll="eventhandler()" pageSize="10" />
    ...
</mx:Application>
```

Sizing a ScrollBar control

The ScrollBar control does not display correctly if it is sized smaller than the height of the up arrow and down arrow buttons. There is no error checking for this condition. Macromedia recommends that you hide the ScrollBar control in such a condition. If there is not enough room for the thumb, the thumb is made invisible.

User interaction

Use the mouse to click the various portions of the ScrollBar control, which broadcasts events to listeners. The object listening to the ScrollBar control is responsible for updating the portion of data displayed. The ScrollBar control updates itself to represent the new state after the action has taken place.

ScrollBar control syntax

You use the `<mx:VScrollBar>` tag to define a vertical ScrollBar control, and the `<mx:HScrollBar>` tag to define a horizontal ScrollBar control. For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*. The following table describes the optional properties defined by these controls:

| Property | Type | Use | Description |
|-----------------------------|--------|----------|---|
| <code>lineScrollSize</code> | Number | Property | Specifies the increment to move when an arrow button is pressed. The default value is 1. |
| <code>maxPos</code> | Number | Property | Specifies the top of the scrolling range. The default value is 0. |
| <code>minPos</code> | Number | Property | Specifies the bottom of the scrolling range. The default value is 0. |
| <code>minWidth</code> | Number | Property | Read-only property that contains the minimum width of the ScrollBar control. |
| <code>minHeight</code> | Number | Property | Read-only property that contains the maximum width of the ScrollBar control. |
| <code>pageScrollSize</code> | Number | Property | Specifies the increment to move when the track is pressed. This value is reset to the <code>pageSize</code> parameter of the <code>setScrollProperties()</code> method when it is called. |
| <code>pageSize</code> | Number | Property | Specifies the page size, the increment to move when the track is pressed, for the ScrollBar control. The default value is 0. |
| <code>scrollPosition</code> | Number | Property | Specifies the current scrolling position. The default value is 0. |
| <code>scroll</code> | | Event | <ul style="list-style-type: none">Specifies a handler for <code>scroll</code> events, which are broadcast when the ScrollBar control state changes. |

Text control

The Text control displays multiline, noneditable text. Noneditable means that the application user cannot modify the text. The Text control does not support scroll bars; its preferred size is a square large enough to display the specified text.

To create a single-line, noneditable text field, use the Label control. For more information, see [“Label control” on page 63](#). To create user-editable text fields, you use the TextInput or TextArea controls. For more information, see [“TextInput control” on page 88](#) and [“TextArea control” on page 85](#).

The following figure shows an example of the Text control:

This is an
example of a
multiline text
string in a Text
control.

The Text control supports HTML text and a variety of text and font styles. The text always word-wraps at the control boundaries, and is always aligned to the top of the control. The Text control is transparent so that the background of the component's container shows through, and the control has no borders.

The Text control has the following default properties:

| Property | Default |
|----------------|---|
| preferred size | A square large enough to contain the text. The width will not be larger than the width of the Application window. |
| minimum size | None |
| maximum size | None |

Creating a Text control

You define a Text control in MXML using the `<mx:Text>` tag, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:Text text="This is an example of a multiline text string in a Text
    control." />
</mx:Application>
```

You use the `text` property to specify a string of raw text, and the `htmlText` property to specify an HTML-formatted string.

Using the text property

You can use the `text` property to specify the text string that appears in the Text control. The control collapses any white-space characters, such as tab and newline characters. Any HTML tags in the text string are ignored, and appear as entered in the string.

For the special characters left angle bracket (`<`), right angle bracket (`>`), and ampersand (`&`), insert the HTML equivalents of `<`, `>`, and `&`. The following example sets the text string using the `text` property:

```
<mx:Text text="This string contains a less than, &lt;, greater than, &gt;, and
  amp, &amp;" />
```

If you wrap the text string in a CDATA tag, you can specify the literal characters for the left angle bracket (<), right angle bracket (>), or ampersand (&) in the text property using a child tag, as the following example shows:

```
<mx:Text >
  <mx:text><![CDATA[This is an example of a multiline text string in a Text
    control with a less than, <, greater than, >, and amp, &. ]]>
  </mx:text>
</mx:Text>
```

The following example uses an initialization function to set the text property to a string that contains these characters:

```
<mx:Script>
  <![CDATA[

    function initText() {
      TA.text="This is an example of a multiline text string in a Text control
        with a less than, <, greater than, >, and amp, &."
    }

  ]]>
</mx:Script>

<mx:Text id="TA" initialize="initText()" />
```

Since the code of the <mx:Script> tag is contained in a CDATA tag, you do not need an additional tag for the string.

Using the htmlText property

You use the htmlText property to specify an HTML-formatted text string. If your text string contains HTML tags, you must wrap it in a CDATA tag. The control collapses any white-space characters, such as tab and newline characters.

When you specify the text string for the Text control in MXML, you cannot escape special characters, such as tab and newline characters. For example, if you include the characters '\n' or '\t' in the text string, the characters appear as '\n' and '\t' in the Text control. To insert tab and newline characters, use the htmlText property and insert the
 tag or escape sequence into the text string.

For the special characters left angle bracket (<), right angle bracket (>), and ampersand (&), insert the HTML equivalents of <, >, and &.

The following example sets the text string to an HTML-formatted string using the htmlText property:

```
<mx:Text>
  <mx:htmlText><![CDATA[This is an example of a multiline text
    string in a Text control, with some carriage returns <br> <br> with a less
    than, &lt;;, greater than, &gt;;, and amp, &amp;.. ]]>
  </mx:htmlText>
</mx:Text>
```

If you omit the `CDATA` tag, Flex converts the `<`, `>`, and `&` back into the literal characters left angle bracket (`<`), right angle bracket (`>`), or ampersand (`&`), and will attempt to interpret them as HTML.

For more information on using HTML tags in `htmlText`, [“Using HTML-formatted text” on page 65](#).

Sizing a Text control

If you do not specify `width` or `height` properties for the Text control, Flex initially sizes the control to be a square large enough to hold the specified text.

If you specify a pixel value for both the `height` and `width` properties, the Text control does not resize in the corresponding direction, and any text that exceeds the size of the control is clipped at the border.

If you specify a pixel value for either the `height` or `width` property but not both, the Text control recalculates the value for the other property and reflows the text to fit. No text is clipped in this case.

If you specify a percentage value for the `width` or `height` property, the Text control stretches in the specified direction to fill the requested percentage of the parent container’s available space if possible.

Note: If you specify a percentage width but don’t specify height, some text may be clipped at certain widths. This happens because Flex sets the fixed height to the size needed by the initial width, which may be less than is needed at other widths.

Text control syntax

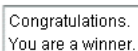
You use the `<mx:Text>` tag to define a Text control. For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*. The Text control is derived from the Label control and accepts all of the properties and methods of the Label control, and the optional property defined in the following table. For more information, see [“Label control” on page 63](#).

| Property | Type | Use | Description |
|-------------------------|---------|----------|---|
| <code>selectable</code> | Boolean | Property | Specifies whether the text can be selected, <code>true</code> , or not, <code>false</code> . Making the text selectable allows you to copy and paste it. The default value is <code>true</code> . |

TextArea control

The TextArea control is a multiline, editable text field with a border and optional scroll bars. All text in a TextArea control must use the same styling unless it is HTML text. The TextArea control supports the HTML rendering capabilities of Flash Player. The TextArea control broadcasts a `change` event.

The following figure shows a TextArea control:



To create a single-line, editable text field, use the `TextInput` control. For more information, see [“TextInput control” on page 88](#).

If you disable a `TextArea` control, it displays its contents in a different color represented by the `disabledColor` style. You can set a `TextArea` control to read-only to disallow editing of the text. You can set a `TextArea` control’s password property to conceal input text by displaying characters as asterisks.

The `TextArea` control has the following default properties:

| Property | Default |
|----------------|--|
| preferred size | Width = 100 pixels; height = 44 pixels |
| minimum size | 0 |
| maximum size | Undefined |

Creating a TextArea control

You define a `TextArea` control in MXML using the `<mx:TextArea>` tag, as the following example shows. Specify an `id` value if you intend to refer to a control elsewhere in your MXML, either in another tag or in an `ActionScript` block.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:TextArea id="textConfirm" text="Congratulations. You are a winner." />
</mx:Application>
```

Just as you can for the `Text` control, you use the `text` property to specify a string of raw text, and the `htmlText` property to specify an HTML formatted string. For more information, see [“Using the text property” on page 83](#) and [“Using the htmlText property” on page 84](#).

TextArea control syntax

You use the `<mx:TextArea>` tag to define a `TextArea` control. For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*. The following table describes the optional properties defined by the `TextArea` control:

| Property | Type | Use | Description |
|----------------------------|---------|----------|--|
| <code>editable</code> | Boolean | Property | Specifies whether the user can edit the text. The default value is <code>true</code> . |
| <code>hPosition</code> | Number | Property | Specifies the pixel position of the leftmost character that is currently displayed. Will always be 0, and ignore changes, if <code>wordWrap</code> is set to <code>true</code> . |
| <code>hScrollPolicy</code> | String | Property | Specifies whether the horizontal scroll bar is always on, <code>on</code> ; always off, <code>off</code> ; or turns on when needed, <code>auto</code> . The default value is <code>auto</code> . |

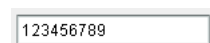
| Property | Type | Use | Description |
|---------------------------|--------|----------|--|
| <code>htmlText</code> | String | Property | <p>Contains HTML formatted text. If your text string contains HTML tags, you must wrap it in a <code>CDATA</code> tag. The control collapses any white-space characters, such as tab and newline characters.</p> <p>For the special characters left angle bracket (<code><</code>), right angle bracket (<code>></code>), ad ampersand (<code>&</code>), insert the HTML equivalents of <code>&lt;</code>, <code>&gt;</code>, and <code>&amp;</code>.</p> <p>The <code>htmlText</code> property ignores CSS style settings for the control, and instead relies on the HTML tags in the string for formatting. Do not try to mix HTML tags and styles; use HTML tags to format your text.</p> |
| <code>length</code> | Number | Property | Read-only property that contains the length of the string. |
| <code>maxChars</code> | Number | Property | Specifies the maximum number of characters that the text field can contain. The default value is <code>undefined</code> . |
| <code>maxHPosition</code> | Number | Property | Specifies the maximum value of <code>hPosition</code> . The default value is 0. Will always be 0 if <code>wordWrap</code> is set to <code>true</code> . |
| <code>maxVPosition</code> | Number | Property | Specifies the maximum value of <code>vPosition</code> . The default value is 0. |
| <code>password</code> | String | Property | <p>Specifies that the field is a password field, <code>true</code>, or not, <code>false</code>. The default value is <code>false</code>.</p> <p>If you set this property to <code>true</code>, each text character entered into the control appears as an asterisk character (*).</p> |
| <code>restrict</code> | String | Property | <p>Specifies the set of characters that a user can enter into the text field. If the value of the <code>restrict</code> property is null or an empty string, you can enter any character.</p> <p>This property only restricts user interaction; a script might put any text into the text field.</p> <p>If the value of the <code>restrict</code> property is a string of characters, you may enter only characters in that string into the text field. The string is scanned from left to right. You can specify a range using the dash (-).</p> <p>If the string begins with a caret (^), the string specifies the characters that cannot be entered into the control. For example, the string <code>"^a-z"</code> means that all uppercase letters may be entered, but no lowercase letters are allowed.</p> <p>This property does not synchronize with the Embed Font Outlines check boxes in the Property inspector.</p> <p>Since some characters have a special meaning when used in the <code>restrict</code> property, you must use the backslash character to specify the literal characters <code>-</code>, <code>^</code>, and <code>\</code>, as follows:</p> <pre>\^ \^- \\</pre> |

| Property | Type | Use | Description |
|----------------------------|--------|----------|---|
| <code>text</code> | String | Property | Specifies the text string that appears in the control. The control collapses any white-space characters, such as tab and newline characters. Any HTML tags in the text string are ignored, and appear as entered in the string. For the special characters left angle bracket (<), right angle bracket (>), and ampersand (&), wrap the text string in a <code>CDATA</code> tag. |
| <code>vPosition</code> | Number | Property | Specifies the line number of the topmost row of characters that is currently displayed. The default value is 0. |
| <code>vScrollPolicy</code> | String | Property | Specifies whether the vertical scroll bar is always on, on, never on, off, or turns on when needed auto. The default value is auto. |
| <code>wordWrap</code> | String | Property | Specifies whether the text wraps, true, or not, false. The default value is true. |
| <code>change</code> | | Event | Broadcast when text in the TextArea control changes by user input or through data binding. This event does not occur if you modify the text using ActionScript code. |
| <code>scroll</code> | | Event | Broadcast when the content is scrolled. |

TextInput control

The TextInput control is a single-line text field that is optionally editable. All text in the TextInput control must use the same styling unless it is HTML text. The TextInput control supports the HTML rendering capabilities of Flash Player. The TextInput control, like other input and choice controls, can have a required value indicator when used in a FormItem container in a Form container.

The following figure shows a TextInput control:



To create a multiline, editable text field, use the TextArea control. For more information, see [“TextArea control” on page 85](#).

TextInput controls do not include a label, but you can add one using a Label control or by nesting the TextInput control in a FormItem container in a Form container. TextInput controls indicate whether a value is required, and have a number of states, including filled, selected, disabled, and error. TextInput controls support formatting, validation, and keyboard equivalents, and broadcast change and enter events.

If you disable a TextInput control, it displays its contents in a different color, represented by the `disabledColor` style. You can set a TextInput control’s editable property to `false` to disallow editing of the text. You can set a TextInput control’s password property to conceal the input text by displaying characters as asterisks.

The TextInput control has the following default properties:

| Property | Default |
|----------------|---|
| preferred size | The size of the text with a default minimum size of 160 pixels. |
| minimum size | 0 |
| maximum size | Undefined |

Creating a TextInput control

You define a TextInput control in MXML using the `<mx:TextInput>` tag, as the following example shows. Specify an `id` value if you intend to refer to a control elsewhere in your MXML, either in another tag or in an ActionScript block.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:TextInput id="text1" width="100"/>
</mx:Application>
```

Just as you can for the Label control, you use the `text` property to specify a string of raw text, and the `htmlText` property to specify an HTML formatted string. For more information, see [“Using the text property” on page 64](#) and [“Using the htmlText property” on page 64](#).

Binding to a TextInput control

In some cases, you might want to bind a variable to the `text` property of a TextInput control, as the following example shows:

```
<mx:TextInput text="{myProp}" />
```

In this example, the TextInput control displays the value of the `myProp` variable. However, if `myProp` has not been initialized, so that its value is null or its value is undefined, the control shows the string `null` or `undefined`. You can configure the control to display an empty string in this case, as the following example shows:

```
<mx:TextInput text="{myProp == null ? '' : myProp}"/>
```

TextInput control syntax

You use the `<mx:TextInput>` tag to define a TextInput control. For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*. The following table describes the optional properties defined by the TextInput control:

| Property | Type | Use | Description |
|------------------------|---------|----------|---|
| <code>editable</code> | Boolean | Property | Specifies whether the user can edit the text. The default value is <code>true</code> . |
| <code>hPosition</code> | Number | Property | Specifies the pixel position of the leftmost character that is currently displayed. The default value is 0. |

| Property | Type | Use | Description |
|---------------------------|---------|----------|---|
| <code>htmlText</code> | String | Property | <p>Contains HTML formatted text. If your text string contains HTML tags, you must wrap it in the <code>CDATA</code> tag. The control collapses any white-space characters, such as tab and newline characters. For the special characters left angle bracket (<code><</code>), right angle bracket (<code>></code>), and ampersand (<code>&</code>), insert the HTML equivalents of <code>&lt;</code>, <code>&gt;</code>, and <code>&amp;</code>.</p> <p>The <code>htmlText</code> property ignores CSS style settings for the control, and instead relies on the HTML tags in the string for formatting. Do not mix HTML tags and styles; use HTML tags to format your text.</p> |
| <code>length</code> | Number | Property | Read-only property that contains the length of the string. |
| <code>maxChars</code> | Number | Property | Specifies the maximum number of characters that the text field can contain. The default value is <code>undefined</code> . |
| <code>maxHPosition</code> | Number | Property | Specifies the maximum value of <code>hPosition</code> . The default value is 0. |
| <code>password</code> | Boolean | Property | <p>Specifies whether the field is a password field, <code>true</code>, or not, <code>false</code>. The default value is <code>false</code>.</p> <p>If you set this property to <code>true</code>, each text character entered into the control appears as the asterisk character (*).</p> |
| <code>restrict</code> | String | Property | Specifies the set of characters that a user can enter into the text field. For more information, see the <code>restrict</code> property of the <code>TextArea</code> control in “TextArea control syntax” on page 86 . |
| <code>text</code> | String | Property | <p>Specifies the text string that appears in the control. The control collapses any white-space characters, such as tab and newline characters.</p> <p>Any HTML tags in the text string are ignored, and appear as entered in the string.</p> <p>For the special characters left angle bracket (<code><</code>), right angle bracket (<code>></code>), and ampersand (<code>&</code>), wrap the text string in a <code>CDATA</code> tag.</p> |
| <code>change</code> | | Event | Specifies a handler for <code>change</code> events, which are broadcast when text in the <code>TextInput</code> control changes. |
| <code>enter</code> | | Event | Specifies a handler for <code>enter</code> events, which are broadcast when the Enter key is pressed. |

CHAPTER 3

Using Data Provider Controls

Several Macromedia Flex controls take input from a data provider. For example, a Tree control reads data from a data provider to define the structure of the tree and any associated data assigned to each tree node.

This chapter describes the data provider controls and includes examples of different ways to populate these controls using a data provider.

Contents

| | |
|----------------------------------|-----|
| About data providers | 91 |
| List control | 105 |
| HorizontalList control | 116 |
| TileList control | 121 |
| DataGrid control | 127 |
| Tree control | 135 |
| ComboBox control | 141 |
| Menu control | 148 |
| MenuBar control | 153 |

About data providers

Several Flex components, such as the Tree and ComboBox controls, take input data from a data provider. A *data provider* is a collection of objects, similar to an Array. For example, a Tree control reads data from a data provider to define the structure of the tree and any associated data assigned to each tree node.

The data provider creates a level of abstraction between Flex components and the data that you use to populate them. You can populate multiple components from the same data provider, switch data providers for a component at runtime, and modify the data provider so that changes are reflected by all components that use the data provider.

You can think of the data provider as the model, and the Flex components as the view onto the model. By separating the model from the view, you can change one without changing the other.

Types of data providers

Flex uses two types of data providers. You use a list-based data provider with the List, DataGrid, ComboBox, TabBar, and LinkBar components. These components use a flat data provider, similar to a one-dimensional Array. For more information on using data providers with these components, see [“Using data providers with list-based components” on page 96](#).

This chapter describes the List, DataGrid, and ComboBox controls. For information on the TabBar and LinkBar containers, see [Chapter 7, “Using Navigator Containers,” on page 247](#).

You use a hierarchical data provider with the Tree, Menu, and MenuBar controls. A hierarchical data provider lets you define a hierarchical data structure that matches the layout of a tree or menu. For example, a tree typically has a root node, with one or more branch or leaf nodes. Each branch node can hold additional branch nodes or leaf nodes, but a leaf node is an endpoint of the tree. For more information on defining data providers for the Tree, Menu, and MenuBar controls, see [“Using data providers with hierarchical controls” on page 101](#).

This chapter also describes the Tree, Menu, and MenuBar controls.

The structure of a data provider

A data provider consists of two parts: a collection of data objects and an API. The data provider API is a set of methods and properties that a class must implement so that a Flex component recognizes it as a data provider.

For a class to function as a list-based data provider, it must implement the data provider interface. The Flex Array class implements this API, so you can use the Array class as a data provider for Flex components. Flash Remoting RecordSets and data from the ActionScript MX 2004 DataSet component also support the data provider API.

For a class to function as a hierarchical data provider, it must implement the TreeDataProvider API. The ActionScript XMLNode class and the Flex TreeNode classes implement this API, so you can use them as data providers for Flex components.

You can also define your own custom classes that implement either data provider API. After you define them, you can use your custom classes as data providers for Flex components.

For a description of the data provider API, see [“Using the data provider API” on page 94](#).

Using a data provider to populate a component

You assign a data provider to a component’s `dataProvider` property. The following example shows a data provider specifying the data for a ComboBox control:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mx" >
  <mx:ComboBox id="myCB" >
    <mx:dataProvider>
      <mx:Array>
        <mx:String>AL</mx:String>
      </mx:Array>
    </mx:dataProvider>
  </mx:ComboBox>
</mx:Application>
```

```

        <mx:String>AK</mx:String>
        <mx:String>AR</mx:String>
    </mx:Array>
</mx:dataProvider>
</mx:ComboBox>
</mx:Application>

```

In this example, you use the `dataProvider` property of the `ComboBox` control to define an Array of String objects.

If each Array element is a simple type, the type's value is displayed as a label in the control. In this example, the value of each String displays in the `ComboBox` control. If each element is an Object, by default the value of the Object's `label` property appears.

The index of items in the data provider Array is zero-based, which means that the values are 0, 1, 2, ..., $n - 1$, where n is the total number of items in the Array. The value of the each data provider item in this example is a text string.

After defining the `ComboBox` control in MXML, the `dataProvider` property of the `ComboBox` control contains a reference to the underlying data provider object. You can modify the data provider in ActionScript using the `dataProvider` property, as the following example shows:

```
myCB.dataProvider.addItem("MA");
```

The data provider API

The data provider API is a set of methods and properties that a class must implement so that a Flex component recognizes it as a data provider. The Flash Remoting `RecordSet` class and ActionScript Array and `DataSet` classes all implement this API.

A data provider for the list- and hierarchical-based components implements the following API:

| List | Hierarchical | Description |
|-------------------------------|------------------------------|---|
| <code>addItem()</code> | <code>addTreeNode()</code> | Adds an item at the end of the data provider. |
| <code>addItemAt()</code> | <code>addTreeNodeAt()</code> | Adds an item to the data provider at the specified position. |
| <code>editField()</code> | | Changes one field of one item of the data provider. |
| | <code>getChildNodes()</code> | Returns all child nodes of the item. |
| | <code>getData()</code> | Returns the data for the entire data provider, not for a specific item. Use <code>getProperty()</code> to access information about a specific item. For example, in an event handler, you can use the following statement: <code>event.target.selectedItem.getProperty("label")</code> |
| <code>getEditingData()</code> | | Gets the data for editing from a data provider. |
| <code>getItemAt()</code> | <code>getTreeNodeAt()</code> | Gets a reference to the item at a specified position. |
| <code>getItemID()</code> | | Returns the unique ID of the item. |
| | <code>getProperty()</code> | Returns a property of the data provider. |
| | <code>hasChildNodes()</code> | Returns <code>true</code> if the item has child nodes. |
| | <code>indexOf()</code> | Returns the index of an item. |

| List | Hierarchical | Description |
|-----------------|--------------------|--|
| removeAll() | removeAll() | Removes all items from a data provider. |
| | removeTreeNode() | Removes the node from its parent in the data provider. |
| removeItemAt() | removeTreeNodeAt() | Removes an item from a data provider at a specified position. |
| replaceItemAt() | | Replaces the item at a specified position with another item. |
| | setData() | Sets the data for an item. |
| | setProperty() | Sets a property of the data provider. |
| sortItems() | | Sorts the items in a data provider. |
| sortItemsBy() | | Sorts the items in a data provider. |
| length | | Property that contains the number of items in a data provider. |
| modelChanged | modelChanged | Event broadcast when the data provider changes. |

Using the data provider API

The data provider API lets you programmatically manipulate the data provider. The following example creates an Array as a data provider for a List control, then populates it at runtime using the API. Because the Array class implements the data provider API, you can use all the methods of the API to manipulate it.

```
<mx:Script>
  <![CDATA[
    // Variable that contains the data provider.
    var myDP : Array;
    // Function to initialize the data provider.
    function initList() {
      myDP = new Array();
      myDP.addItem("one");
      myDP.addItem("two");
      myDP.addItem("three");
      myList.dataProvider = myDP;
    }
  ]]>
</mx:Script>
```

```
<mx:List id="myList" initialize="initList()" />
```

The last statement of the function initializes the List control by passing a reference to the data provider Array to the List control. Therefore, if you use the data provider API to manipulate the Array after initializing the List control, those changes are propagated to the List control. Also, if you use the same data provider for multiple controls, any changes to the data provider propagate to all controls.

The following example uses the `addItem()` method of the data provider API to add a new item to the data provider in response to some event:

```
function addToList(newEntry) {  
    myDP.addItem(newEntry);  
}
```

You can specify a data type to the `newEntry` argument based on the data type of the Array elements in your data provider.

The Array class implements the data provider API, so you can call the `addItem()` method directly on the Array object. The Array class signals modifications to any component that uses it so that the component updates as well.

Because the `dataProvider` property of the List control contains a reference to its underlying data provider object, you can also implement the `addToList()` function by calling methods on the `dataProvider` property, as the following example shows:

```
function addToList(newEntry) {  
    myList.dataProvider.addItem(newEntry);  
}
```

These two implementations of the `addToList()` function are equivalent.

You can change the data provider associated with a component at runtime. The following example changes the data provider of a DataGrid control in response to a user action:

```
function changedDP(newDP) {  
    myList.dataProvider = newDP;  
}
```

Using the data provider API of Flex components

Individual Flex components define an API that components use to communicate with their data provider. You can use the component's API to modify the data provider in the same way that you can use the data provider API directly.

The following example rewrites the `addToList()` function from the previous section to use methods of the List object, rather than of the data provider:

```
function addToList(newEntry:String) {  
    myList.addItem(newEntry);  
}
```

Any changes you make to a data provider using the component's API propagate to any other components that also use that data provider. That is, each component maintains a reference to its data provider; it does not store its own local copy. Therefore, if in the previous example, the List control shared its data provider with another List control, the new entry would appear in both controls.

Converting an object to an Array for use with a data providers

Controls that use data providers require that the data provider contains an Array. In some cases, you might populate a data provider from an external data source using a Flex remote object service, HTTP service, or web service. If you are sure that the data service returns an Array, you bind its results to the data provider of the control, as the following example shows:

```
<mx:ComboBox>
  <mx:dataProvider>{myResults}</mx:dataProvider>
</mx:ComboBox>
```

For more information and examples on binding to a data provider, see [“Passing data to a ComboBox control” on page 98](#). For more information on data services, see [Chapter 28, “Managing Data in Flex,” on page 629](#).

For data downloaded using the Flex HTTPService or represented by the `<mx:Model>` tag, if Flex encounters a single tag in the data, Flex treats it as a single object. If Flex encounters multiple copies of the same tag, Flex treats it as an Array.

If you are unsure whether the result of the data service contains an Array or an individual object, you can use the `toArray()` method of the `mx.utils.ArrayUtil` class to convert it to an Array. If you pass the `toArray()` method to an individual object, it returns an Array with that object as the only Array element. If you pass it an Array, it returns the same Array.

The following example uses the `toArray()` method as part of the binding statement:

```
<mx:ComboBox>
  <mx:dataProvider>{mx.utils.ArrayUtil.toArray(myResults)}</mx:dataProvider>
</mx:ComboBox>
```

You usually do not need to use the `toArray()` method for web services, because there is a schema associated with the web service that specifies to Flex whether an item is an Array, even if it contains only a single object.

Defining a custom data provider

You can define your own data providers to populate Flex controls. This gives you the opportunity to build your own logic into the data provider to have total control over how Flex displays your data.

A custom data provider gives you a great deal of control over the data that populates the associated control. For example, you might have a data provider that uses a calculation to determine the data to return to the associated control.

Using data providers with list-based components

Flex list-based components use a data provider to supply data to the component. These components include the following:

- ComboBox control
- DataGrid control
- List control

- LinkBar container
- TabBar container

The list-based components all use a similar mechanism with the data provider. The examples in this section use the ComboBox control, but apply to the other components.

The LinkBar and TabBar containers have an additional option that is not available with the other controls. For more information, see [“Using data providers with LinkBar and TabBar containers” on page 100.](#)

Using a data provider in MXML

You specify the data for the ComboBox control using the `<mx:dataProvider>` child tag of the `<mx:ComboBox>` tag. The `<mx:dataProvider>` tag lets you specify data in several ways. In the simplest case for creating a ComboBox control, you use the `<mx:dataProvider>`, `<mx:Array>`, and `<mx:String>` tags to define the entries as an Array of Strings, as the following example shows:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:ComboBox>
    <mx:dataProvider>
      <mx:Array>
        <mx:String>AL</mx:String>
        <mx:String>AK</mx:String>
        <mx:String>AR</mx:String>
      </mx:Array>
    </mx:dataProvider>
  </mx:ComboBox>
</mx:Application>
```

Using objects to populate a ComboBox control

You can populate a ComboBox control with an Array of Objects. Objects let you define a `label` property that contains the string displayed in the ComboBox control, and a `data` property that contains any data that you want to associate with the label, as the following example shows:

```
<mx:ComboBox>
  <mx:dataProvider>
    <mx:Array>
      <mx:Object label="AL" data="Montgomery"/>
      <mx:Object label="AK" data="Juneau"/>
      <mx:Object label="AR" data="Little Rock"/>
    </mx:Array>
  </mx:dataProvider>
</mx:ComboBox>
```

The `label` property contains the state name and the `data` property contains the name of the state's capital. By default, the ComboBox control uses the `label` property of each Object in the data provider to determine the text displayed in the control.

If each Object does not contain a `label` property, you can use the `labelField` property to specify the property name, as the following example shows:

```
<mx:ComboBox labelField="state" >
  <mx:dataProvider>
    <mx:Array>
      <mx:Object state="Alabama" data="Montgomery"/>
      <mx:Object state="Alaska" data="Juneau"/>
      <mx:Object state="Arkansas" data="Little Rock"/>
    </mx:Array>
  </mx:dataProvider>
</mx:ComboBox>
```

You can also use a property name other than `data` for additional information.

Passing data to a ComboBox control

Flex lets you populate the data provider of a ComboBox control from an ActionScript variable definition or from a Flex data model. When you use a variable, you can define it so that each Array element contains one of the following:

- A label (string)
- A label (string) paired with data (scalar value or Object)

The following example populates a ComboBox control from a variable:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

  <mx:Script>
    <![CDATA[
      var COLOR_ARRAY:Array=
        [{label:"Red", data:"#FF0000"},
         {label:"Green", data:"#00FF00"},
         {label:"Blue", data:"#0000FF"}];
    ]]>
  </mx:Script>

  <mx:ComboBox >
    <mx:dataProvider>
      {COLOR_ARRAY}
    </mx:dataProvider>
  </mx:ComboBox>
</mx:Application>
```

Since you can also specify the `dataProvider` property as a property of the `<mx:CombBox>` tag, you can write the definition of the ComboBox control in the previous example as the following code shows:

```
<mx:ComboBox dataProvider = "{COLOR_ARRAY}" />
```

You can also bind a Flex data model to the `<mx:dataProvider>` property, as the following example shows:

```
<mx:Model id="myDP">
  <obj>
    <item label="AL" data="Montgomery"/>
    <item>
      <label>AK</label>
      <data>Juneau</data>
    </item>
    <item>
      <label>AR</label>
      <data>Little Rock</data>
    </item>
  </obj>
</mx:Model>

<mx:ComboBox dataProvider="{myDP.obj.item}" />
```

In this example, you populate the `ComboBox` control with the `item` Array from the model. This example uses a simple model. However, you can populate the model from an external data source or define a custom data model class in ActionScript. For more information on using data models, see [Chapter 28, “Managing Data in Flex,” on page 629](#).

You can bind a web service operation’s result object to the `dataProvider` property of an `<mx:ComboBox>` tag. For example, when a web service operation returns an Array of strings, you can use the following syntax to display each string as a row of a `ComboBox` control:

```
<mx:ComboBox dataProvider="{service.operation.result}" />
```

For more information on using web services, see [Chapter 30, “Using Data Services,” on page 655](#).

Manipulating a list-based data provider at runtime

The following example puts together many of the data provider concepts described in previous sections. In this example, you create an Array that defines the data provider of a `ComboBox` control, and then use the data provider API to manipulate the data provider:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

  <mx:Script>
    <![CDATA[
      var COLOR_ARRAY:Array=
        [{label:"Red", data:"#FF0000"},
         {label:"Green", data:"#00FF00"},
         {label:"Blue", data:"#0000FF"}
        ];

      function addColor() {
        combo2.dataProvider.addItem("SMG", "#FF00FF");
      }
    ]]>
  </mx:Script>

  <mx:ComboBox id="combo2" dataProvider="{COLOR_ARRAY}" />
</mx:Application>
```

```

        function removeColor() {
            combo2.dataProvider.removeItemAt(3);
        }
    ]]>
</mx:Script>

<mx:ComboBox id="combo2" dataProvider="{COLOR_ARRAY}" />
<mx:Button label="Add Color" click="addColor()" />
<mx:Button label="Remove Color" click="removeColor()" />
</mx:Application>

```

Using data providers with LinkBar and TabBar containers

The LinkBar and TabBar containers use data providers in a similar way as do ComboBox, DataGrid, and List controls. The difference is that the `dataProvider` property of LinkBar and TabBar containers can be one of three different types:

Data provider array The LinkBar and TabBar containers behave the same as the ComboBox, DataGrid and List controls, and the syntax is the same as for using a data provider with those controls.

ViewStack identifier One of the most common uses of a LinkBar or TabBar container is to control the active child of a ViewStack container, as the following example shows:

```

<!-- Create a LinkBar container to hold the three links. -->
<mx:LinkBar dataProvider="myViewStack" borderStyle="solid" />
<!-- Define the ViewStack and the three child containers. -->
<mx:ViewStack id="myViewStack" borderStyle="solid" width="100%">
    <mx:Canvas id="search" label="Search">
        <mx:Label text="Search Screen" />
    </mx:Canvas>
    <mx:Canvas id="custInfo" label="Customer Info">
        <mx:Label text="Customer Info" />
    </mx:Canvas>
    <mx:Canvas id="accountInfo" label="Account Info">
        <mx:Label text="Account Info" />
    </mx:Canvas>
</mx:ViewStack>

```

In this example, the links of the LinkBar container display the `label` name of the child containers of the ViewStack container. For more information, see [“ViewStack navigator container” on page 248](#).

String If you specify a string to the data provider, Flex interprets it as an identifier. Flex searches for an object with the specified identifier in the current scope. The identifier can reference a ViewStack container, or an Array; for example:

```

<mx:Array id="myArray">
    <mx:String>One</mx:String>
    <mx:String>Two</mx:String>
</mx:Array>

<mx:TabBar dataProvider="myArray" />

```

Using data providers with hierarchical controls

A data provider for the hierarchical controls defines a nested hierarchy of nodes and subnodes. The following Flex components use a hierarchical data provider:

- Tree control
- Menu control
- MenuBar control

The hierarchical components all use the same mechanism to work with the data provider. The examples in this section use the Tree control but also apply to the other components also.

Using a hierarchical data provider in MXML

You define a Tree control in MXML using the `<mx:Tree>` tag. The data provider of a Tree control must support the `TreeDataProvider` API. Often, the data for a tree is retrieved from a server in the form of XML, but it can also be well-formed XML defined within the `<mx:Tree>` tag.

The following code example defines a Tree control:

```
<mx:Tree id="myTree" >
  <mx:dataProvider>
    <mx:XML>
      <node label="Mail">
        <node label="INBOX"/>
        <node label="Personal Folder">
          <node label="Business" />
          <node label="Demo" />
          <node label="Personal" isBranch="true" />
          <node label="Saved Mail" />
        </node>
        <node label="Sent" />
        <node label="Trash"/>
      </node>
    </mx:XML>
  </mx:dataProvider>
</mx:Tree>
```

Flex parses the `<mx:XML>` tag to create an XML Object that implements the `TreeDataProvider` API, much in the same way that `Array` implements the `DataProvider` API.

There are many ways to structure XML. Flex controls are not designed to use all types of XML structures, so it is important to use XML that the controls can interpret. Do not nest node properties in a child node; each node should contain all its necessary properties and attributes. Also, the properties of each node should be consistent to be useful. For example, to describe a mailbox structure with a Tree component, use the same properties on each node (message, data, time, attachments, and so on). This lets the Tree control anticipate which it will render, and lets you loop through the hierarchy to compare data.

Node tags in the XML data can have any name. Notice in the previous example that each node is named with the generic `<node>` tag. Flex controls read the data provider and build the display hierarchy based on the nesting relationship of the nodes.

Nodes at the highest level are called *root* nodes and have no parent. Controls can have multiple root nodes. In this example, there is only one root node in the tree: “Mail”. However, if you added sibling nodes at that level, multiple root nodes would be displayed in the Tree. Branch nodes can contain multiple child nodes. Leaf nodes cannot contain child nodes.

When a Tree control displays a node, it displays the `label` property of the node by default as the text label.

Passing XML data to a Tree control

You can build your XML tree structure within a Flex data model, as the following example shows:

```
<mx:XML id="myDP">
  <node label="Mail">
    <node label="INBOX"/>
    <node label="Personal Folder">
      <node label="Business" />
      <node label="Demo" />
      <node label="Personal" isBranch="true" />
      <node label="Saved Mail" />
    </node>
    <node label="Sent" />
    <node label="Trash"/>
  </node>
</mx:XML>

<mx:Tree id="myTree">
  <mx:dataProvider>
    {myDP}
  </mx:dataProvider>
</mx:Tree>
```

In this example, you define your data provider using the Flex `<mx:XML>` tag, and then use that tag as the data provider for your Tree control. By defining your Tree hierarchy in a data model, you can populate the model at runtime. For example, you can use a web service to define the Tree control. For more information on data models, [Chapter 28, “Managing Data in Flex,” on page 629](#).

Manipulating a hierarchical data provider at runtime

The data provider for the hierarchical controls supports the API defined by the Flex `TreeDataProvider` interface. Therefore, you can use all of the properties and methods of this interface to manipulate the data provider. If you define your own data provider, your class must implement the `TreeDataProvider` API. For more information, see [“The data provider API” on page 93](#).

The data provider of a hierarchical component defines a hierarchy of nodes, and each node of the hierarchy supports the `TreeDataProvider` API. Therefore, you use the data provider API to manipulate the hierarchy, and every node in it.

The following example defines a Tree control:

```
<mx:Tree id="myTree" width="300" height="500" >
  <mx:dataProvider>
    <mx:XML>
      <node label="Mail">
        <node label="INBOX"/>
        <node label="Personal Folder">
          <node label="Business" data="2"/>
          <node label="Demo" />
          <node label="Personal" isBranch="true" />
          <node label="Saved Mail" />
        </node>
        <node label="Sent" />
        <node label="Trash"/>
      </node>
    </mx:XML>
  </mx:dataProvider>
</mx:Tree>
```

To create a new node at the root level of the hierarchy, you use the `TreeDataProvider` `addTreeNode()` method:

```
function addNode(){
  // Add a root tree node at index 1 with a label of SMG,
  // and a data value of 1.
  myTree.dataProvider.addTreeNodeAt(1, "SMG", "1");
}
```

To add a node to a subnode, you use the `TreeDataProvider` API to traverse the hierarchy to obtain the node, then modify it. The `getTreeNodeAt()` and `getTreeNode()` methods of the `TreeDataProvider` API return an object that supports the `TreeDataProvider` API.

The following example adds a node to the first subnode of the first root node of the XML tree:

```
function addChildNode(){
  // Get first root node.
  var node0=tree1.dataProvider.getTreeNodeAt(0);
  // Get first subnode of the root node.
  var node00=node0.getTreeNodeAt(0);
  // Add new node.
  node00.addTreeNodeAt(0, "SMG", "1");
}
```

Creating a data provider using the `TreeNode` class

You can also create your data model programmatically using the `TreeDataProvider` API and the `TreeNode` class. The `TreeNode` class implements the `TreeDataProvider` API, as the following example shows:

```
<mx:Script>
  <![CDATA[
    // Import the TreeNode.
    import mx.controls.treeclasses.TreeNode;
    // Create a data provider variable.
    var treeDP;
```

```

function initTree(){
    // Populate the data provider variable with the root node,
    // and two subnodes.
    treeDP = new TreeNode();
    var root = treeDP.addTreeNode("root","0");
    root.addTreeNode("node 1 ","1");
    root.addTreeNode("node 2","2");

    // Create a third subnode, and two subnodes of it.
    var node3 = root.addTreeNode("node 3","3");
    node3.addTreeNode("node 3-0", "3-0");
    node3.addTreeNode("node 3-1", "3-1");

    // Initialize the tree data provider.
    myTree.dataProvider = treeDP;
}
]]>
</mx:Script>

<mx:Tree id="myTree" initialize="initTree()" />

```

In this example, you use the `addTreeNode()` method of the `TreeDataProvider` API to create nodes. This method returns a node object that you can manipulate. The node object also supports the `TreeDataProvider` API so that you can build your data provider hierarchy.

Passing an object as the data provider

You can pass an object that contains a data structure to a hierarchical control. Flex automatically wraps the object in a `TreeNode` object so that you can use the `TreeDataProvider` API to manipulate it.

The following example creates an `Array`, then passes it to the `Tree` control:

```

<mx:Script>
    <![CDATA[
        import mx.controls.treeclasses.TreeNode;
        var treeDP:Array = [
            {node1: {label:"node11"}, node11: {label:"node11"}},
            {node2: {label:"node21"}, node21: {label:"node22"}}];
        function initTree(){
            myTree.dataProvider = treeDP;
        }
    ]]>
</mx:Script>

<mx:Tree id="myTree" initialize="initTree()" />

```

Flex wraps the object as a `TreeNode` object, so you manipulate the object as you would any `TreeDataProvider`. The following example adds a new node to the data provider:

```
myTree.dataProvider.addTreeNodeAt(1, "SMG", "1");
```


Considerations when using hierarchical data providers

A hierarchical control can take as its data provider a deserialized object tree defined by an `<mx:Model>` tag, or an object tree assigned directly to the `dataProvider` property. Flex wraps the tree in a `TreeNode` object so that you can use the `TreeDataProvider` API to manipulate the tree.

You should be aware of the following considerations:

- If the object tree is created from an `<mx:Model>` tag or deserialized XML received from a server, Flex generates nodes for child objects in the order they appeared in the original XML.
- If a child object in an object tree is an `Array`, its values become child nodes of the `Array`'s parent object. If the `Array` is the top-level item in the object tree, each `Array` element becomes a root-level node.
- Any child nodes added to or removed from a `TreeNode` that represents an object tree are not reflected in the underlying data object. Changes to properties, made using the `setProperty()` and `getProperty()` methods, are reflected in the underlying data object and the associated control.

List control

The List control displays a vertical list of single-line items. Its functionality is very similar to that of the `SELECT` form element in HTML. It usually contains a vertical scroll bar that lets users access the items in the list. An optional horizontal scroll bar lets users view items when the full width of the list items is unlikely to fit. The user can select one or more items from the list.

The `DataGrid` control and `Tree` control inherit all the properties and methods of the List control.

The following figure shows a List control:



The List control has the following default properties:

| Property | Default |
|----------------|---|
| preferred size | 200 pixels wide and seven rows high, where each row is 20 pixels high |
| minimum size | 30 pixels wide and three rows high |
| maximum size | undefined |

Creating a List control

You use the `<mx:List>` tag to define a List control. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an `ActionScript` block.

The List control uses a list-based data provider. For more information, see [“Using data providers with list-based components” on page 96](#).

You specify the data for the List control using the `<mx:dataProvider>` child tag of the `<mx:List>` tag. The `<mx:dataProvider>` tag lets you specify data in several different ways. In the simplest case for creating a List control, you use the `<mx:dataProvider>`, `<mx:Array>`, and `<mx:String>` tags to define the entries as an Array of Strings, as the following example shows:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

    <mx:List>
        <mx:dataProvider>
            <mx:Array>
                <mx:String>AK</mx:String>
                <mx:String>AL</mx:String>
                <mx:String>AR</mx:String>
            </mx:Array>
        </mx:dataProvider>
    </mx:List>
</mx:Application>
```

The index of items in the List control is zero-based, which means that values are 0, 1, 2, ..., $n - 1$, where n is the total number of items. The value of the item is its label text.

You typically use events to handle user interaction with a List control. The following example code adds a handler for a change event to the List control. Flex broadcasts a change event when the value of the control changes due to user interaction.

```
<mx:Script>
    <![CDATA[
        function changeEvt(event) {
            forChange.text=event.target.selectedItem + " " + " " +
                event.target.selectedIndex;
        }
    ]]>
</mx:Script>

<mx:List change="changeEvt(event)" >
    ...
</mx:List>
<mx:TextArea id="forChange" width="150" />
```

In this example, you use two properties of the List control, `selectedItem` and `selectedIndex`, in the event handler. Every change event updates the TextArea control with the label of the selected item and the item's index in the control.

The `target` property of the object passed to the event handler contains a reference to the List control. You can reference any control property using `target`. The `target.selectedItem` field contains a copy of the selected item. If you populate the List control with an Array of Strings, the `target.selectedItem` field contains a String. If you populate it with an Array of Objects, the `target.selectedItem` field contains the Object that corresponds to the selected item.

Using a label function

You can pass a label function to the List control to provide logic that determines the text that appears in the control. The following example uses a function to combine the values of the `label` and `data` fields for each item for display in the List control:

```
<mx:Script>
    
        function myLabelFunc(item):String {
            return item.data + ", " + item.label;
        }
    ]]&gt;
&lt;/mx:Script&gt;

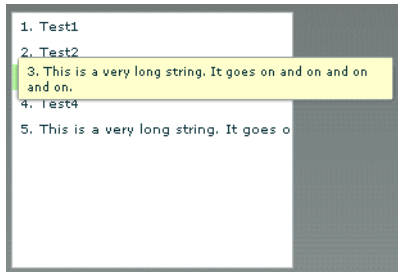
&lt;mx>List labelFunction="myLabelFunc" &gt;
    &lt;mx:dataProvider&gt;
        &lt;mx:Array&gt;
            &lt;mx:Object label="AL" data="Montgomery"/&gt;
            &lt;mx:Object label="AK" data="Juneau"/&gt;
            &lt;mx:Object label="AR" data="Little Rock"/&gt;
        &lt;/mx:Array&gt;
    &lt;/mx:dataProvider&gt;
&lt;/mx>List&gt;</pre></div><div data-bbox="144 410 895 448" data-label="Text"><p>The function takes a single argument, <code>item</code>, which contains the list item and returns the String to display in the List control. This example creates the following List control:</p></div><div data-bbox="151 464 300 542" data-label="Image"><img alt="A screenshot of a List control displaying three items: 'Montgomery, AL', 'Juneau, AK', and 'Little Rock, AR'. The text is left-aligned and stacked vertically within a rectangular border." data-bbox="151 464 300 542"/></div><div data-bbox="144 551 892 570" data-label="Text"><p>Note that for the DataGrid control, which is a subclass of the List class, the method signature is:</p></div><div data-bbox="144 576 664 593" data-label="Text"><pre>labelFunction(item:Object, columnName:String) : String</pre></div><div data-bbox="144 600 842 636" data-label="Text"><p>where <code>item</code> contains the DataGrid item object, and <code>columnName</code> contains the name of the DataGrid column.</p></div><div data-bbox="125 653 334 673" data-label="Section-Header"><h2>Displaying DataTips</h2></div><div data-bbox="144 681 895 774" data-label="Text"><p>DataTips are similar to ToolTips, but display text when the mouse pointer hovers over a row in a List control. To display DataTips, you set the <code>showDataTips</code> property of a List control to <code>true</code>. Text in the rows of a List control is often longer than the width of the List control, and is clipped on the right side. DataTips can solve that problem by displaying all of the text, including the clipped text, when the mouse pointer hovers over a cell.</p></div><div data-bbox="144 781 848 813" data-label="Text"><p><b>Note:</b> To use DataTips with a DataGrid control, you must set the <code>showDataTips</code> property on the individual DataGridColumn of the DataGrid.</p></div><div data-bbox="752 936 895 953" data-label="Page-Footer"><hr/><p>List control 107</p></div>
```

The default behavior of the `showDataTips` property is to display the full contents of the text in the row. However, you can use the `dataTipField` and `dataTipFunction` properties to determine what is displayed in the DataTip. The `dataTipField` property behaves like the `labelField` property; it specifies the name of the field in the data provider to use as the DataTip field for cells in the column. The `dataTipFunction` property behaves like the `labelFunction` property; it specifies the DataTip string to display for list items.

The following example sets the `showDataTips` property for a List control:

```
<mx:List id="myList" dataProvider="{myDP}" width="220" height="200"
  showDataTips="true" />
```

This example creates the following List control:



Displaying ScrollTips

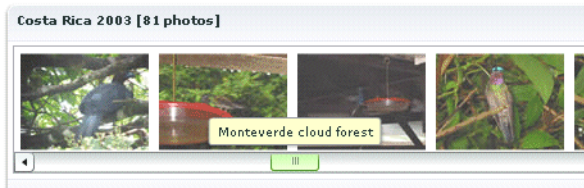
The default value is `false`. You use `ScrollTips` to give users context about where they are in the list as they scroll through the list. The default value of the `showScrollTips` property is `false`.

The default behavior of the `showScrollTips` property is to display the index number of list items. You can use the `scrollTipFunction` property to determine what is displayed in the ScrollTip. The `scrollTipFunction` property behaves like the `labelFunction` property; it specifies the ScrollTip string to display for list items. You should avoid going to the server to fill in a ScrollTip.

The following example sets the `showScrollTips` and `scrollTipFunction` properties of a `HorizontalList` control. The `scrollTipFunction` property specifies a function that gets the value of the `description` property of the current list item.

```
<mx:HorizontalList id="list" dataProvider="{album.photo}" width="100%"
  cellRenderer="Thumbnail" itemWidth="108" height="100"
  selectionColor="#FFCC00" liveScrolling="false" showScrollTips="true"
  scrollTipFunction="scrollTipFunc"
  change="currentPhoto=album.photo[list.selectedIndex]"/>
```

This example creates the following `HorizontalList` control:



Note: The Photo Viewer application included in the `samples.war` file contains the code used in this example. You can extract the `samples.war` file to your application server.

Vertically aligning text in List control rows

You can use the `verticalAlignment` property to vertically align text at the top, center, or bottom of a List row. The default value is `top`. You can also specify a value of `center` or `bottom`.

The following example sets the `verticalAlignment` property for a List control to `center`:

```
<mx:List id="myList" dataProvider="{myDP}" width="220" height="200"
  verticalAlignment="center" />
```

Setting variable height for List control rows

You can use the `variableRowHeight` property to make the height of List control rows variable based on their content. The default value is `false`. If you set the `variableRowHeight` property to `true`, the `rowHeight` property is ignored and the `rowCount` property is read-only.

The following example sets the `variableRowHeight` property for a List control to `true`:

```
<mx:List id="myList" dataProvider="{myDP}" width="220" height="200"
  variableRowHeight="true" />
```

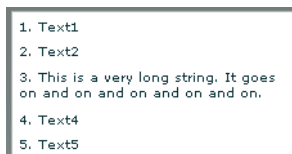
Wrapping text in List control rows

You can use the `wordWrap` property in combination with the `variableRowHeight` property to wrap text to multiple lines when it exceeds the width of a List row.

The following example sets the `wordWrap` and `variableRowHeight` properties to `true`:

```
<mx:List id="myList" dataProvider="{myDP}" width="220" height="200"
  variableRowHeight="true" wordWrap="true" />
```

This MXML code creates the following List control:



Note: To use the `wordWrap` property with a `DataGrid` control, you must define `DataGridColumn`s explicitly, set `wordWrap="true"` on the `<mx:DataGridColumn>` tag, and set `variableRowHeight="true"` on the `<mx:DataGrid>` tag.

Using a custom cell renderer

A cell renderer is the object that displays a List control's data items. The simplest way to use a custom cell renderer is to specify an MXML component as the value of the `cellRenderer` property. When you use an MXML component as a cell renderer, it can contain multiple levels of containers and controls. The MXML component must implement the `setVaLue()` method. You can also use an ActionScript class as a custom cell renderer. For more information, see [“Creating a cell renderer” on page 379](#).

The following example sets the `cellRenderer` property to an MXML component named `FancyCellRenderer`. It also sets the `variableRowHeight` property to `true` because the MXML component exceeds the default row height:

```
<mx:List id="myList1" dataProvider="{myDP}" width="220" height="200"
  cellRenderer="{FancyCellRenderer}" variableRowHeight="true"/>
```

Sorting a list

You can use methods of the List control to sort the list items. The following example uses the `sortItemsBy()` method to sort the list in descending order, based on the label, in response to a Button click event:

```
<mx:Script>
  <![CDATA[
    function sortList(evt) {
      myList.sortItemsBy("label", "DESC");
    }
  ]]>
</mx:Script>

<mx:List id="myList" >
  ...
</mx:List>
<mx:Button label="Sort" click="sortList()" />
```

The `sortItemsBy()` method can take the name of any field of the list item to use as the sorting field. For example, if you include a data field for each list item, you can sort on that field, as the following example shows:

```
<mx:Script>
  <![CDATA[
    function sortList(evt) {
      myList.sortItemsBy("data", "DESC");
    }
  ]]>
</mx:Script>

<mx:List >
  <mx:dataProvider>
    <mx:Array>
      <mx:Object label="AL" data="Montgomery"/>
      <mx:Object label="AK" data="Juneau"/>
      <mx:Object label="AR" data="Little Rock"/>
    </mx:Array>
  </mx:dataProvider>
</mx:List>
```

```

    </mx:dataProvider>
</mx:List>
<mx:Button label="Sort" click="sortList()" />

```

Specifying an icon to the List control

You can specify an icon to display with each List item, as the following example shows:

```

<mx:Script>
    <![CDATA[
        [Embed(source="yahoosmall.jpg")]
        var iconSymbol1:String;
        [Embed(source="atom.jpg")]
        var iconSymbol2:String;
    ]]>
</mx:Script>

<mx:List iconField="myIcon" >
    <mx:dataProvider>
        <mx:Array>
            <mx:Object label="AL" data="Montgomery" myIcon="iconSymbol1" />
            <mx:Object label="AK" data="Juneau" myIcon="iconSymbol2" />
            <mx:Object label="AR" data="Little Rock" myIcon="iconSymbol1" />
        </mx:Array>
    </mx:dataProvider>
</mx:List>

```

In this example, you use the `iconField` property to specify the field of each item containing the icon. You use the `Embed` metadata to import the icons, and then reference them in the List control definition.

You can also use the `iconFunction` property to specify a function that determines the icon, much in the same way that you can use the `labelFunction` property to specify a function that determines the label text. The following example shows a List control that uses the `iconFunction` property to determine the icon to display for each item in the list:

```

<mx:Script>
    <![CDATA[
        // Embed icons.
        [Embed("PavementIcon.jpg")]
        var pavementSymbol:String;
        [Embed("NormalIcon.jpg")]
        var normalSymbol:String;

        // Define data provider.
        var myDP : Array;
        function initList() {
            myDP = [
                { Artist:'Pavement', Album:'Slanted and Enchanted', Price:11.99 },
                { Artist:'Pavarotti', Album:'Twilight', Price:11.99 },
                { Artist:'Other', Album:'Other', Price:5.99 } ];

            list1.dataProvider = myDP;
        }
    ]]>
</mx:Script>

```

```

        // Determine icon based on artist. Pavement gets a special icon.
        function myiconfunction( item ){
            var type:String = item.Artist;
            if (type == "Pavement") {
                return pavementSymbol;
            }
            return normalSymbol;
        }
    ]]>
</mx:Script>

<mx:VBox >
    <mx:List id="list1" initialize="initList()" labelField="Artist"
        iconFunction="myiconfunction" />
</mx:VBox>

```

Alternating row colors in a List control

You can use the `alternatingRowColors` style property to specify an Array that defines the color of each row in the List control. The Array must contain two or more colors. After using all the entries in the Array, the List control repeats the color scheme.

The following example defines an Array with two entries, `#FF0000` for red and `#00FF00` for green. Therefore, the rows of the List control alternate between these two colors.

```
<mx:List alternatingRowColors=["#FF0000", "#00FF00"].../ >
```

User interaction

The user clicks individual list items to select them, and holds down the Control and Shift keys while clicking to select multiple items.

All mouse or keyboard selections broadcast a `change` event. For mouse interactions, the List control broadcasts this event when the mouse button is released.

When the user drags the mouse over the rows and then outside the control, the control scrolls up or down.

A List control shows one less than the number of records that fit in the display. Paging down through a 10-line list shows records 0-9, 9-18, 18-27, and so on, with one line overlapping from one page to the next.

Keyboard navigation

The List control has the following keyboard navigation features:

| Key | Action |
|------------|--------------------------------|
| Up arrow | Moves selection up one item. |
| Down arrow | Moves selection down one item. |
| Page Up | Moves selection up one page. |
| Page Down | Moves selection down one page. |

| Key | Action |
|-------------------|---|
| Home | Moves selection to the top of the list. |
| End | Moves selection to the bottom of the list. |
| Alphanumeric keys | Jumps to the next item that begins with the character typed. |
| Control | Toggle key. Allows for multiple (noncontiguous) selection and deselection. Works with key presses, click selection, and drag selection. |
| Shift | Contiguous selection key. Allows for contiguous selections. Works with key presses, click selection, and drag selection. |

List control syntax

You use the `<mx:List>` tag to define a List control. For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*. The following table describes the optional properties defined by the List control:

| Property | Type | Use | Description |
|------------------------------|----------|----------|---|
| <code>cellRenderer</code> | String | Property | Specifies the class object reference or symbol linkage ID of the cell renderer to use. For more information, see Chapter 15, “Customizing Data Provider Controls,” on page 379 . |
| <code>dataProvider</code> | Array | Property | Specifies an Array of simple types or objects to populate the List control. |
| <code>dataTipField</code> | String | Property | Specifies the name of the field in the data provider to use as the DataTip field for list items. |
| <code>dataTipFunction</code> | Function | Property | Specifies a function that determines the DataTip string to display for list items. The callback function takes a single argument, which contains the data for the entire row, and returns a String. The signature of the callback function is: <i>mydataTipFunc(item:Object):String</i> |
| <code>hPosition</code> | Number | Property | Specifies the pixel position of the horizontal scroll bar, where a value of 0 corresponds to the left end of the scroll bar. The default value is 0. |
| <code>hScrollPolicy</code> | String | Property | Specifies when to include a horizontal scroll bar. A value of <code>on</code> causes the container to always include a horizontal scroll bar; a value of <code>off</code> (default) always excludes it. |
| <code>iconField</code> | String | Property | Specifies a field within each item to be used as a way of specifying icons. Takes the value of the field and uses it as an icon identifier. If the field has a value of <code>undefined</code> , the default icon is used. |
| <code>iconFunction</code> | Function | Property | Specifies a function used to determine which icon to use. This function receives a single argument, <code>item</code> , which is the item being rendered, and must return a string representing the <code>iconID</code> to use. For an example, see “Specifying an icon to the List control” on page 111 . |

| Property | Type | Use | Description |
|--------------------------------|----------|----------|--|
| <code>labelField</code> | String | Property | Specifies the name of the field that the objects of the data provider use as the label field. If omitted, the data provider must contain a field named <code>label</code> , or the <code>dataProvider</code> property must contain an Array of Strings. |
| <code>labelFunction</code> | Function | Property | Specifies a function that determines the content to display for each list item. This function must return a String representing the text to display. For the List control, and all subclasses of List except the DataGrid control, the method signature of the function is: <code>labelFunction(item:Object) : String</code> where <code>item</code> contains the list item object. For the DataGrid control, the method signature is: <code>labelFunction(item:Object, columnName:String) : String</code> where <code>item</code> contains the DataGrid item object, and <code>columnName</code> contains the name of the DataGrid column. For an example, see “Using a label function” on page 107 . |
| <code>liveScrolling</code> | Boolean | Property | Determines whether scrolling is live as the user moves the thumb or the view is not updated until the user releases the thumb. If <code>true</code> , scrolling is live. The default value is <code>true</code> . By setting the <code>liveScrolling</code> property to <code>false</code> , you can improve the performance of a List for which rendering of cells requires the loading of non-embedded images, there are significant changes in the layout of the cell, or you are using a data provider for which the data is not fully loaded and must undergo additional calculations or retrieval from the server. |
| <code>maxHPosition</code> | Number | Property | Specifies the number of pixels to the right that the List control can scroll when <code>hScrollPolicy</code> has been set to <code>on</code> . The list does not precisely measure the width of text within it. |
| <code>multipleSelection</code> | Boolean | Property | Specifies whether multiple selection of items is allowed, <code>true</code> , or not, <code>false</code> (default). |
| <code>rowCount</code> | Number | Property | Specifies the maximum number of rows visible in the List control. The default value is based on the height of the text. |
| <code>rowHeight</code> | Number | Property | Specifies the pixel height of every row in the List control. The font settings do not grow the List rows to fit, so <code>rowHeight</code> is the best way to ensure items completely fit. The default value is 20 pixels. Changing <code>rowHeight</code> does not affect the height of the control, but does affect the number of visible rows, as reported by <code>rowCount</code> . |
| <code>rowRenderer</code> | String | Property | Specifies the class object reference or symbol linkage ID of the row renderer to use. The default value is <code>SelectableRow</code> . |

| Property | Type | Use | Description |
|--------------------------------|----------|----------|---|
| <code>scrollTipFunction</code> | Function | Property | <p>Specifies a function that is called if the <code>showScrollTips</code> property is set to <code>true</code> and the scroll thumb is being dragged and should return the String to be used as a ScrollTip. The method is passed two parameters. The first parameter is the direction of the scroll bar. The second parameter is its <code>scrollPosition</code>. The following example shows a <code>scrollTipFunction</code> function:</p> <pre>function scrollTipFunc (dir : String, pos: Number) : String { return album.photo[pos].description; }</pre> |
| <code>selectedIndex</code> | Number | Property | <p>Contains the index of the selected item.</p> <p>Setting this property in ActionScript sets the current index, and selects the associated label in the List control. The default value is <code>undefined</code>, which means that no item is selected.</p> |
| <code>selectedIndices</code> | Array | Property | <p>Contains the indices of multiple selected items in the list. If you click the second item, the third item, and then the first item, <code>selectedIndices</code> contains <code>[1,2,0]</code>.</p> <p>Setting this property replaces the current selection. Setting it to 0 clears the current selection.</p> |
| <code>selectedItem</code> | Object | Property | <p>For a single-selection control, it contains the selected item. In a multiple-selection control, it contains the most recently selected item.</p> <p>The value can be a single scalar value or an object containing <code>label</code> and <code>data</code> properties. This is a read-only property.</p> |
| <code>selectedItems</code> | Array | Property | <p>For a multiple-selection control, it contains the set of selected items.</p> <p>Items are in the order of selection. If you click the second item, the third item, and then the first item, <code>selectedItems</code> contains <code>[1,2,0]</code>.</p> <p>The value of each item can either be a single scalar value or an object containing <code>label</code> and <code>data</code> properties.</p> |
| <code>showDataTips</code> | Boolean | Property | <p>Specifies whether a column of cells shows DataTips, <code>true</code>, or not, <code>false</code>. The default value is <code>false</code>. DataTips are like ToolTips, but are for individual cells in DataGridColumn and List controls.</p> <p>To use DataTips with a DataGrid control, you must set the <code>showDataTips</code> property on the individual DataGridColumn of the DataGrid control.</p> |
| <code>showScrollTips</code> | Boolean | Property | <p>Specifies whether a ScrollTip should appear near the scroll thumb when it is being dragged. The default value is <code>false</code>. You use ScrollTips to give users context about where they are in the list.</p> |

| Property | Type | Use | Description |
|--------------------------------|---------|----------|---|
| <code>variableRowHeight</code> | Boolean | Property | Specifies whether the height of rows is variable. The default value is <code>false</code> . If <code>true</code> , the <code>rowHeight</code> property is ignored and the <code>rowCount</code> property is read-only. |
| <code>verticalAlignment</code> | String | Property | Specifies the vertical alignment of row content. The possible values are <code>top</code> (default), <code>center</code> , and <code>bottom</code> . |
| <code>vPosition</code> | Number | Property | Sets the topmost visible item of the list. If you set this property to an index number that doesn't exist, the list scrolls to the nearest index. The default value is 0. |
| <code>vScrollPolicy</code> | String | Property | Specifies when to include a vertical scroll bar. A value of <code>on</code> (default) causes the container to always include a vertical scroll bar; a value of <code>off</code> always excludes it. |
| <code>wordWrap</code> | Boolean | Property | Specifies whether list items wrap words to the next line, <code>true</code> , or not, <code>false</code> . The default value is <code>false</code> . To use the <code>wordWrap</code> property with a <code>DataGrid</code> control you must define <code>DataGridColumn</code> s explicitly, and then set <code>wordWrap="true"</code> on the <code><mx:DataGridColumn></code> tag and set <code>variableRowHeight="true"</code> on the <code><mx:DataGrid></code> tag. |
| <code>change</code> | | Event | Specifies a handler for <code>change</code> events, which are broadcast when the value of the control changes as a result of user interaction. |
| <code>itemRollOver</code> | | Event | Specifies a handler for <code>itemRollOver</code> events, which are broadcast when the user rolls over items. |
| <code>itemRollOut</code> | | Event | Specifies a handler for <code>itemRollOut</code> events, which are broadcast when the user rolls out of list items. |
| <code>scroll</code> | | Event | Specifies a handler for <code>scroll</code> events, which are broadcast when the user scrolls through the list. |

HorizontalList control

The `HorizontalList` control displays a horizontal list of items. The `HorizontalList` control is particularly useful in combination with a custom cell renderer for displaying a list of images and other data. For more information about custom cell renderers, see [Chapter 15, “Customizing Data Provider Controls,”](#) on page 379.

The contents of a `HorizontalList` control can look very similar to the contents of an `HBox` container in which a `Repeater` object repeats components. However, performance of a `HorizontalList` control can be better than the combination of an `HBox` container and a `Repeater` object because the `HorizontalList` control only instantiates the objects that fit in its display area. Scrolling in a `HorizontalList` can be slower than it is when using a `Repeater` object. For more information about the `Repeater` object, see [Chapter 8, “Dynamically Repeating Controls and Containers,”](#) on page 269.

The `HorizontalList` control always displays items from left to right. The control usually contains a horizontal scroll bar, which lets users access all items in the list. An optional vertical scroll bar lets users view items when the full height of the list items is unlikely to fit. The user can select one or more items from the list, depending on the value of the `multipleSelection` property.

The following figure shows a `HorizontalList` control:



The `HorizontalList` control has the following default properties:

| Property | Default |
|----------------|---|
| preferred size | Seven columns, each of which is 80 pixels wide and 80 pixels high; total width 560 pixels |
| minimum size | Three columns; 240 pixels wide and about 20 pixels high |
| maximum size | Undefined |

Creating a `HorizontalList` control

You use the `<mx:HorizontalList>` tag to define a `HorizontalList` control. The `HorizontalList` control has the properties specified in [“HorizontalList control syntax” on page 120](#) and most of the properties and methods of the `List` control. For more information about using the `List` control, see [“List control” on page 105](#). Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The `HorizontalList` control uses a list-based data provider. For more information, see [“Using data providers with list-based components” on page 96](#).

You specify the data for a `HorizontalList` control using the `dataProvider` property of the `<mx:HorizontalList>` tag.

Note: The Flex Explorer application included in the `samples.war` file contains the applications used in the following examples. You can extract the `samples.war` file to your application server.

The following example shows the MXML code for an application that displays a catalog of product images in a `HorizontalList` control. The cell renderer for the `HorizontalList` control is an MXML component named `Thumbnail`.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:Model id="catalog" source="../assets/catalog.xml"/>

  <mx:HorizontalList
    dataProvider="{catalog.product}"
    width="100%"
```

```

        cellRenderer="Thumbnail"
        itemWidth="120"
        height="125"/>

<mx:Link label="Product images courtesy of Lavish"
        click="getUrl('http://www.shoplavish.com', '_blank')"/>
</mx:Application>

```

The following example shows the Thumbnail.mxml MXML component that is used as the cell renderer in the product catalog application. The component implements a `setValue()` method, which is required for cell renderers, and contains Image and Label controls. For more information about custom cell renderers, see [Chapter 15, “Customizing Data Provider Controls,” on page 379](#).

```

<?xml version="1.0" ?>
<mx:VBox xmlns:mx="http://www.macromedia.com/2003/mxml"
        horizontalAlign="center"
        verticalGap="0" borderStyle="none">

    <mx:Script>
        var product: Object;
        function setValue(str: String, item: Object) {
            if (item==undefined) {
                visible = false;
                return;
            } else {
                product=item;
                visible=true;
            }
        }
    </mx:Script>

    <mx:Image id="image" width="60" height="60" source="{product.image}"/>
    <mx:Label text="{product.name}" width="120" textAlign="center"/>
    <mx:Label text="{product.price}" fontWeight="bold"/>
</mx:VBox>

```

The following example shows the code for a slightly more complex application that uses a **HorizontalList** control:

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
        xmlns="*"
        backgroundColor="#FFFFFF"
        horizontalAlign="left"
        initialize="initApp()">
    ...
    <mx:Script>
        <![CDATA[
            var albumName: String="album/album.xml";
            // The album XML node deserialized into an object graph by the
            HTTPService
            var album: Object;
            ...
            // reference to the currently displayed photo object
            var currentPhoto: Object;

```

```

function initApp() {
    albumSrv.send();
}

function albumSrvResult(event) {
    album=albumSrv.result.album;
    photoCount=album.photo.length;
    list.selectedIndex=0;
    currentPhoto=album.photo[list.selectedIndex];
}

...
function scrollTipFunc(dir : String, pos: Number) : String {
    return album.photo[pos].description;
}
]]>
</mx:Script>

<mx:HTTPService id="albumSrv" url="{albumName}"
    result="albumSrvResult(event)"/>

<mx:HBox width="100%" height="100%">
    ...
    <mx:VRule height="100%"/>
    <mx:VBox verticalGap="2" width="100%" height="100%"
        visible="{photoCount>0}">
        <mx:Label text="{album.title}"/>
        <mx:Label text="Photo {list.selectedIndex+1} of {photoCount}"/>
        <mx:Text text="{currentPhoto.description}" width="100%"
            height="100%"/>
    </mx:VBox>
</mx:HBox>

<mx:Panel width="100%" verticalGap="0" title="{album.title}"
    [{photoCount} photos]>

    <mx:VBox width="100%" height="100%">
        <mx:HorizontalList id="list"
            dataProvider="{album.photo}"
            width="100%"
            cellRenderer="Thumbnail"
            itemWidth="108"
            height="100"
            selectionColor="#FFCC00"
            liveScrolling="false"
            showScrollTips="true"
            scrollTipFunction="scrollTipFunc"
            change="currentPhoto=album.photo[list.selectedIndex]"/>
    </mx:VBox>
    ...
</mx:Panel>
</mx:Application>

```

This photo viewer application uses a data service to populate a `HorizontalList` control with a set of thumbnail photos. The `liveScrolling` property of the `HorizontalList` control is set to `false`. Also, the `showScrollTips` property is set to `true`, and the `scrollTipFunction` property is set to a function named `scrollTipFunc` that is defined in the Script tag. The application displays a full-size image based on the thumbnail image that is currently selected in the `HorizontalList` control. The cell renderer for the `HorizontalList` control is an MXML component named `Thumbnail.mxml`. This application gets its data from an data service result; for more information about data services, see [Chapter 30, “Using Data Services,” on page 655](#).

User interaction

The user clicks individual list items to select them, and holds down the Control and Shift keys while clicking to select multiple items.

All mouse or keyboard selections broadcast a `change` event. For mouse interactions, the `HorizontalList` control broadcasts this event when the mouse button is released. When the user drags the mouse over the items and then outside the control, the control scrolls up or down.

A `HorizontalList` control shows one less than the number of records that fit in the display. Paging down through a ten-line list shows records 0-9, 9-18, 18-27, and so on, with one line overlapping from one page to the next.

Keyboard navigation

The `HorizontalList` control has the following keyboard navigation features:

| Key | Action |
|------------|---|
| Page Up | Moves selection to the left one page. |
| Left arrow | Moves selection to the left one item. |
| Down arrow | Moves selection down one item. |
| Page Down | Moves selection to the right one page. |
| Home | Moves selection to the beginning of the list. |
| End | Moves selection to the end of the list. |
| Control | Toggle key. Allows for multiple (noncontiguous) selection and deselection when the <code>multipleSelection</code> property is set to <code>true</code> . Works with key presses, click selection, and drag selection. |
| Shift | Contiguous selection key. Allows for contiguous selections when <code>multipleSelection</code> is set to <code>true</code> . Works with key presses, click selection, and drag selection. |

HorizontalList control syntax

You use the `<mx:HorizontalList>` tag to define a `HorizontalList` control. For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*.

The `HorizontalList` control has almost all of the properties and methods of the `List` control, as described in [“List control syntax” on page 113](#). The following table lists properties specific to the `HorizontalList` control, some of which serve similar purposes as properties of the `List` control but have different names:

| Property | Type | Use | Description |
|---------------------------|-------------------|----------|--|
| <code>columnCount</code> | Number | Property | The number of columns displayed in the control. If this property is not set explicitly, it is computed based on the width of the control and the value of the <code>itemWidth</code> property. |
| <code>itemWidth</code> | Number | Property | The width of each item. The list does not resize items automatically. It determines item size from the value of the <code>itemWidth</code> property. |
| <code>itemRenderer</code> | String/ Object | Property | Specifies the class object reference or symbol linkage ID of the item renderer to use. The default value is <code>SelectableItem</code> . This property is similar to the <code>rowRenderer</code> property of the <code>List</code> control. |
| <code>sampleItem</code> | Object | Property | A sample item found in the <code>dataProvider</code> . The <code>HorizontalList</code> control uses this value to determine the size of individual list items. Flex creates an instance of the specified object and stores it with the control instead of waiting for it come back as part of a larger <code>dataProvider</code> that gets its data from a data service. A sample item typically represents a maximum value so that each cell in the list is sized large enough to fit all of its content. When using a cell renderer with a hard-coded size, you can use a dummy value and not worry about it being the maximum size. |

Note: The `HorizontalList` and `TileList` controls are inconsistent with the `List` and `DataGrid` controls in style propagation to cells. The `HorizontalList` and `TileList` controls do not pass styles specified in the `HorizontalList` or `TileList` tag down to their cells. The `List` and `DataGrid` controls do pass some of these properties down to their cells.

For more information about the `HorizontalList` control API, see *Flex ActionScript and MXML API Reference*.

TileList control

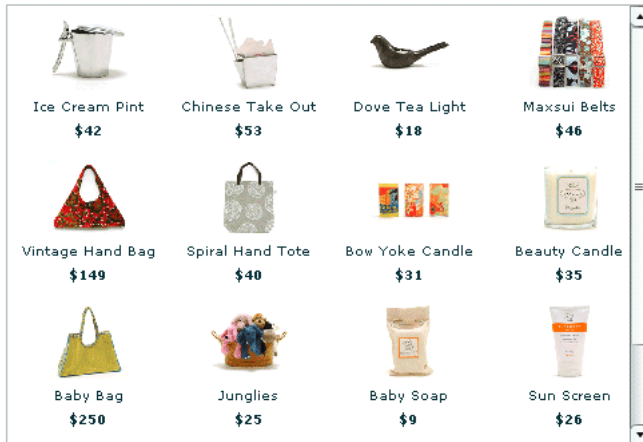
The `TileList` control displays a tiled list of items. The items are tiled in vertical columns or horizontal rows. The `TileList` control is particularly useful in combination with a custom cell renderer for displaying a list of images and other data. For more information about custom cell renderers, see [Chapter 15, “Customizing Data Provider Controls,” on page 379](#).

The contents of a `TileList` control can look very similar to the contents of a `Tile` container in which a `Repeater` object repeats components. However, performance of a `TileList` control can be better than the combination of a `Tile` container and a `Repeater` object because the `TileList` control only instantiates the objects that fit in its display area. Scrolling in a `TileList` can be slower than it is when using a `Repeater` object. For more information about the `Repeater` object, see [Chapter 8, “Dynamically Repeating Controls and Containers,” on page 269](#).

The `TileList` control displays a number of items laid out in equally sized tiles. It usually contains a scroll bar on one of its axes to access all items in the list depending on the direction. The user can select one or more items from the list depending on the value of the `multipleSelection` property.

The `TileList` control lays out its children in one or more vertical columns or horizontal rows, starting new rows or columns as necessary. The `direction` property determines the layout. The possible values for the `direction` property are `vertical` for a column layout and `horizontal` (default) for a row layout.

The following figure shows a `TileList` control:



The `TileList` control has the following default properties:

| Property | Default |
|----------------|---|
| preferred size | Four columns and four rows of 80x80 pixel cells; total size is 320x320 pixels |
| minimum size | One column and one row; total size is 80x80 pixels |
| maximum size | Undefined |

Creating a `TileList` control

You use the `<mx:TileList>` tag to define a `TileList` control. The `TileList` control has the properties specified in “[TileList control syntax](#)” on page 126 and most of the properties and methods of the `List` control. For more information about using the `List` control, see “[List control](#)” on page 105. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an `ActionScript` block.

The `TileList` control uses a list-based data provider. For more information, see “[Using data providers with list-based components](#)” on page 96.

You specify the data for a `TileList` control using the `dataProvider` property of the `<mx:TileList>` tag.

Note: The Flex Explorer application included in the samples.war file contains the applications used in the following examples. You can extract the samples.war file to your application server.

The following example shows the MXML code for a catalog application that displays a set of product images in a TileList control. The cell renderer for the TileList control is an MXML component named Thumbnail. For more information about custom cell renderers, see [Chapter 15, “Customizing Data Provider Controls,” on page 379](#).

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">

    <mx:Model id="catalog" source="../../../assets/catalog.xml"/>

    <mx:TileList
        dataProvider="{catalog.product}"
        width="100%"
        height="100%"
        cellRenderer="Thumbnail"
        itemWidth="120"
        itemHeight="108"/>

    <mx:Link label="Product images courtesy of Lavish" click="getUrl('http://
www.shoplavish.com', '_blank')"/>
</mx:Application>
```

The following example shows the Thumbnail.mxml MXML component that is used as the cell renderer in the product catalog application. The component implements a `setValue()` method, which is required for cell renderers, and contains Image and Label controls. For more information about custom cell renderers, see [Chapter 15, “Customizing Data Provider Controls,” on page 379](#).

```
<?xml version="1.0" ?>
<mx:VBox xmlns:mx="http://www.macromedia.com/2003/mxml"
    horizontalAlign="center"
    verticalGap="0" borderStyle="none">

    <mx:Script>
        var product: Object;
        function setValue(str: String, item: Object) {
            if (item==undefined) {
                visible = false;
                return;
            } else {
                product=item;
                visible=true;
            }
        }
    </mx:Script>

    <mx:Image id="image" width="60" height="60" source="{product.image}"/>
    <mx:Label text="{product.name}" width="120" textAlign="center"/>
    <mx:Label text="{product.price}" fontWeight="bold"/>
</mx:VBox>
```

The following example shows the code for a slightly more complex application that uses a `TileList` control:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
  xmlns="*"
  backgroundColor="#FFFFFF"
  horizontalAlign="left"
  initialize="initApp()">
  ...
  <mx:Script>
    <![CDATA[
      var albumName: String="album/album.xml";

      // The album XML node deserialized into an object graph by the
      HTTPService
      var album: Object;
      ...
      // reference to the currently displayed photo object
      var currentPhoto: Object;

      function initApp() {
        albumSrv.send();
      }

      function albumSrvResult(event) {
        album=albumSrv.result.album;
        photoCount=album.photo.length;
        list.selectedIndex=0;
        currentPhoto=album.photo[list.selectedIndex];
      }
      ...
    ]]>
  </mx:Script>

  <mx:HTTPService id="albumSrv" url="{albumName}"
    result="albumSrvResult(event)"/>

  <mx:HBox width="100%" height="100%">
    ...
    <mx:VRule height="100%" />
    <mx:VBox verticalGap="2" width="100%" height="100%"
      visible="{photoCount>0}">
      <mx:Label text="{album.title}" />
      <mx:Label text="Photo {list.selectedIndex+1} of {photoCount}" />
      <mx:Text text="{currentPhoto.description}" width="100%"
        height="100%" />
    </mx:VBox>
  </mx:HBox>

  <mx:Panel width="100%" verticalGap="0" title="{album.title}"
    [{photoCount} photos]>

    <mx:VBox width="100%" height="100%">
      <mx:TileList id="list">
```

```

        dataProvider="{album.photo}"
        width="100%"
        height="100%"
        cellRenderer="Thumbnail"
        itemWidth="108"
        selectionColor="#FFCC00"
        change="currentPhoto=album.photo[list.selectedIndex]"/>
    </mx:VBox>
    ...
</mx:Panel>
</mx:Application>

```

This photo viewer application uses a data service to populate a `TileList` control with a set of thumbnail photos. The application displays a full-size image based on the thumbnail image that is currently selected in the `TileList` control. The cell renderer for the `TileList` control is an MXML component named `Thumbnail.mxml`. This application gets its data from an data service result; for more information about data services, see [Chapter 30, “Using Data Services,” on page 655](#).

User interaction

The user clicks individual list items to select them, and holds down the Control and Shift keys while clicking to select multiple items.

All mouse or keyboard selections broadcast a `change` event. For mouse interactions, the `TileList` control broadcasts this event when the mouse button is released. When the user drags the mouse over the items and then outside the control, the control scrolls up or down.

Keyboard navigation

The `TileList` control has the following keyboard navigation features:

| Key | Action |
|-------------|--|
| Up arrow | Moves selection up one item. |
| Right arrow | Moves selection to the right one item. |
| Left arrow | Moves selection to the left one item. |
| Down arrow | Moves selection down one item. |
| Page Up | Moves selection up one page. |
| Page Down | Moves selection down one page. |
| Home | Moves selection to the beginning of the list. |
| End | Moves selection to the end of the list. |
| Control | Toggle key. Allows for multiple (noncontiguous) selection and deselection when <code>multipleSelection</code> is set to <code>true</code> . Works with key presses, click selection, and drag selection. |
| Shift | Contiguous selection key. Allows for contiguous selections when <code>multipleSelection</code> is set to <code>true</code> . Works with key presses, click selection, and drag selection. |

TileList control syntax

The TileList control has almost all the same properties and methods as the List control, as described in “[List control syntax](#)” on page 113. For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*. The properties listed in the following table serve similar purposes as properties of the List control but have different names:

| Property | Type | Use | Description |
|--------------|-------------------|----------|---|
| columnCount | Number | Property | The number of columns displayed in the control. If this property is not set explicitly, it is computed based on the width of the control and the <code>itemWidth</code> value. |
| direction | String | Property | Specify which direction the Tile will wrap: vertical means that subsequent items go down and then right, horizontal means that subsequent items go right, and then down. |
| itemHeight | Number | Property | Specifies the pixel height of every item in the control. The default value is 80 pixels. The list does not resize items automatically; it determines item size from the values of the <code>itemWidth</code> and <code>itemHeight</code> properties. This property is similar to the <code>rowHeight</code> property of the List control. |
| itemWidth | Number | Property | The width of each item. The default value is 80 pixels. The list does not resize items automatically; it determines item size from the values of the <code>itemWidth</code> and <code>itemHeight</code> properties. The List control does not have a similar property. |
| itemRenderer | String/ Object | Property | Specifies the class object reference or symbol linkage ID of the item renderer to use. The default value is <code>SelectableItem</code> . This property is similar to the <code>rowRenderer</code> property of the List control. |
| sampleItem | Object | Property | A sample item found in the <code>dataProvider</code> . The control uses this value to determine the size of individual list items. Flex creates an instance of the specified object and stores it with the control instead of waiting for it to come back as part of a larger <code>dataProvider</code> that gets its data from a data service. A sample item typically represents a maximum value so that each cell in the list is sized large enough to fit all of its content. When using a cell renderer with a hard-coded size, you can use a dummy value and not worry about it being the maximum size. |

Note: The HorizontalList and TileList controls are inconsistent with the List and DataGrid controls in style propagation to cells. The HorizontalList and TileList controls do not pass styles specified in the HorizontalList or TileList tag down to their cells. The List and DataGrid controls do pass some of these properties down to their cells.

For more information about the TileList control API, see *Flex ActionScript and MXML API Reference*.

DataGrid control

The DataGrid control is a list that can display more than one column of data. It is a formatted table of data that lets you set editable table cells, and is the foundation of many data-driven applications.

The DataGrid control provides the following features:

- Resizable, sortable, and customizable column layouts
- Optional customizable column and row headers
- Columns that the user can resize at runtime
- Multiple modes of selection (row, column, cell, and edit) and selection events
- Ability to use a custom cell renderer for any column
- Support for paging through data

The following figure shows a DataGrid control:

| Artist | Album | Price |
|----------|----------------------------|-------|
| Pavement | Slanted and Enchanted | 11.99 |
| Pavement | Crooked Rain, Crooked Rain | 10.99 |
| Pavement | Wowee Zowee | 12.99 |
| Pavement | Brighten the Corners | 11.99 |
| Pavement | Terror Twilight | 11.99 |
| Other | Other | 5.99 |

Rows are responsible for rendering items. Each row is laid out vertically below the previous one. Columns are responsible for maintaining the state of each visual column; columns control width, color, and size.

The DataGrid control has the following default properties:

| Property | Default |
|----------------|--|
| preferred size | If the columns are empty, the default width is 300 pixels. If the columns contain information but define no explicit widths, the default width is 100 pixels per column. The default number of rows displayed is 10. |
| minimum size | 0 |
| maximum size | undefined |

Creating a DataGrid control

You use the `<mx:DataGrid>` tag to define a DataGrid control in MXML. The DataGrid control is derived from the List control and takes all of the properties and methods of the List control; for more information about using the List control, see [“List control” on page 105](#). Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The DataGrid control uses a list-based data provider. For more information, see [“Using data providers with list-based components” on page 96](#).

You specify the data for the DataGrid control using the `<mx:dataProvider>` child tag of the `<mx:DataGrid>` tag. The `<mx:dataProvider>` tag lets you specify data in several different ways. In the simplest case for creating a DataGrid control, you use the `<mx:dataProvider>`, `<mx:Array>`, and `<mx:Object>` tags to define the entries as an Array of Objects. Each Array Object defines a row of the DataGrid control, and properties of the Object define the column entries for the row, as the following example shows:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:DataGrid>
    <mx:dataProvider>
      <mx:Array>
        <mx:Object>
          <Artist>Pavement</Artist>
          <Price>11.99</Price>
          <Album>Slanted and Enchanted</Album>
        </mx:Object>
        <mx:Object>
          <Artist>Pavement</Artist>
          <Album>Brighten the Corners</Album>
          <Price>11.99</Price>
        </mx:Object>
      </mx:Array>
    </mx:dataProvider>
  </mx:DataGrid>
</mx:Application>
```

The column names displayed in the DataGrid control are the property names of the Array Objects. By default, the order of columns in the Data Grid control is determined by the order in which you define the properties in the first Array Object. Subsequent objects can define their properties in the same order, or in a different order. If an Array Object omits a property, the DataGrid control displays an empty cell in that row.

The previous example defined the objects using child tags. You can also define the objects using properties, as the following example shows:

```
<mx:DataGrid>
  <mx:dataProvider>
    <mx:Array>
      <mx:Object Artist="Pavement" Price="11.99"
        Album="Slanted and Enchanted" />
      <mx:Object Artist="Pavement"
        Album="Brighten the Corners" Price="11.99" />
    </mx:Array>
  </mx:dataProvider>
</mx:DataGrid>
```

Specifying the column order

You use the `columns` property of the DataGrid control and the `<mx:DataGridColumn>` tag to select the columns to display, specify the order in which to display them, and set additional properties. For more information on the `<mx:DataGridColumn>` tag, see [“DataGridColumn syntax” on page 134](#).

You specify an Array to the `<mx:columns>` child tag of the `<mx:DataGrid>` tag, as the following example shows:

```
<mx:DataGrid>
  <mx:dataProvider>
    <mx:Array>
      <mx:Object Artist="Pavement" Price="11.99"
        Album="Slanted and Enchanted" />
      <mx:Object Artist="Pavement"
        Album="Brighten the Corners" Price="11.99" />
    </mx:Array>
  </mx:dataProvider>
  <mx:columns>
    <mx:Array>
      <mx:DataGridColumn columnName="Album" />
      <mx:DataGridColumn columnName="Price" />
    </mx:Array>
  </mx:columns>
</mx:DataGrid>
```

In this example, you only display the Album and Price columns in the DataGrid control. You can reorder the columns as well, as the following example shows:

```
<mx:columns>
  <mx:Array>
    <mx:DataGridColumn columnName="Price" />
    <mx:DataGridColumn columnName="Album" />
  </mx:Array>
</mx:columns>
```

In this example, you specify that the Price column is the first column in the DataGrid control, and that the Album column is the second.

You can also use the `<mx:DataGridColumn>` tag to set other options. The following example uses the `headerText` property to set the name of the column to a value different than the default name of Album:

```
<mx:columns>
  <mx:Array>
    <mx:DataGridColumn columnName="Price" />
    <mx:DataGridColumn columnName="Album" headerText="Record" />
  </mx:Array>
</mx:columns>
```

Passing data to a DataGrid control

Flex lets you populate a DataGrid control from an ActionScript variable definition or from a Flex data model. The following example populates a DataGrid control from a variable:

DataGrid control:

```
<mx:Script>
  <![CDATA[
    var initDG:Array = [
      { Artist:'Pavement', Album:'Slanted and Enchanted', Price:11.99 },
      { Artist:'Pavement', Album:'Brighten the Corners', Price:11.99 }];
```

```

]]>
</mx:Script>

<mx:DataGrid id="myGrid" width="350" height="350" dataProvider="{initDG}" >
  <mx:columns>
    <mx:Array>
      <mx:DataGridColumn columnName="Album" />
      <mx:DataGridColumn columnName="Price" />
    </mx:Array>
  </mx:columns>
</mx:DataGrid>

```

In this example, you bind the variable `initDG` to the `<mx:dataProvider>` property. You can still specify a column definition event when using data binding.

Limitation when binding data to a DataGrid control

If you use data binding to set the first element of the data provider of a DataGrid control, the associated column does not display. The following example shows a DataGrid control definition using this type of data binding:

```

<mx:DataGrid >
  <mx:dataProvider>
    <mx:Array>
      <mx:Object Album="{myAlbum}" price="{myPrice}" mediaType="CD" />
    </mx:Array>
  </mx:dataProvider>
</mx:DataGrid>

```

In this example, the column corresponding to the `Album` property does not display.

To work around this, use the following syntax to initialize the DataGrid control:

```

<mx:Array id="myArray">
  <mx:Object Album="{myAlbum}" price="{myPrice}" mediaType="CD" />
</mx:Array>

<mx:DataGrid dataProvider="{myArray}" />

```

Handling events in a DataGrid control

The DataGrid control defines several different event types that let you respond to user interaction. For example, Flex broadcasts the `cellPress` event when a user selects an item in a DataGrid control. You can handle this event as the following example shows:

```

<mx:Script>
  <![CDATA[
    function cellPressEvt(event){
      focusCol.text=event.columnIndex;
      focusItem.text=event.itemIndex;
      focusType.text=event.type;
    }
  ]]>
</mx:Script>

<mx:VBox>

```

```

<mx:DataGrid id="myGrid" width="350" height="350"
    cellPress="cellPressEvt(event);" >
    <mx:dataProvider>
        <mx:Array>
            <mx:Object Artist="Pavement" Price="11.99"
                Album="Slanted and Enchanted" />
            <mx:Object Artist="Pavement" Album="Brighten the Corners"
                Price="11.99" />
        </mx:Array>
    </mx:dataProvider>
</mx:DataGrid>

<mx:TextArea id="focusCol" />
<mx:TextArea id="focusItem" />
<mx:TextArea id="focusType" />
</mx:VBox>

```

In this example, you use the event handler to display the column index, item index, and event type in three `TextArea` controls.

The index of columns in the `DataGrid` control is zero-based, meaning values are 0, 1, 2, ..., $n - 1$, where n is the total number of columns. Row items are also indexed starting at 0. Therefore, if you select the first item in the second row, this example displays 0 in the first `Text Area` control for the column index, and 1 in the second `TextArea` control for the item index in the column.

To access the selected item in the event handler, you can use the `target` property of the event object, and the `selectedItem` property of the `DataGrid` control, as the following code shows:

```
var selectedArtist:String=event.target.selectedItem.Artist;
```

The `target` property of the object passed to the event handler contains a reference to the `DataGrid` control. You can reference any control property using `target`. The `target.selectedItem` field contains the selected item. If you populate the `DataGrid` control with an `Array` of `Strings`, the `target.selectedItem` field contains a string. If you populate it with an `Array` of `Objects`, the `target.selectedItem` field contains the object corresponding to the selected item.

User interaction

The `DataGrid` control responds to mouse and keyboard activity. The response to a mouse click or key press depends on whether a cell is editable. A cell is editable when the `editable` property of the `DataGrid` control and the `DataGridColumn` containing the cell are both set to `true`.

If the value of the `sortableColumns` property is `true`, the default value, clicking within a column header causes the `DataGrid` control to be sorted based on the column's cell values. If the value of the `resizableColumns` property is `true`, the default value, clicking in the area between columns permits column resizing.

Clicking within an editable cell directs focus to that cell. Clicking a noneditable cell has no effect on the focus.

Keyboard navigation

The DataGrid control has the following keyboard navigation features:

| Key | Action |
|----------------------------|---|
| Enter, Return, Shift+Enter | When a cell is in editing state, commits change, and moves editing to the cell on the same column, next row down or up, depending on whether Shift is pressed. |
| Tab | Moves focus to the next editable cell, traversing the cells in row order. If at the end of the last row, advances to the next element in the parent container that can receive focus. |
| Shift+Tab | Moves focus to the previous item. If at the beginning of a row, advances to the end of the previous row. If at the beginning of the first row, advances to the previous element in the parent container that can receive focus. |
| Up arrow | If currently editing a cell, shifts the cursor to the beginning of the cell's text. If the cell is not editable, moves selection up one item. |
| Down arrow | If currently editing a cell, shifts the cursor to the end of the cell's text. If the cell is not editable, moves selection down one item. |
| Control | Toggle key. Allows for multiple (noncontiguous) selection and deselection. Works with key presses, click selection, and drag selection. |
| Shift | Contiguous select key. Allows for contiguous selections. Works with key presses, click selection, and drag selection. |

DataGrid control syntax

You use the `<mx:DataGrid>` tag to define a DataGrid control. The DataGrid control inherits the properties and methods of the List control, as described in [“List control syntax” on page 113](#). The DataGrid control also defines the optional properties described in the following table. If an inherited property or method has functionality specific to the DataGrid control, it is included in the table. For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*.

Note: The `themeColor` style does not apply to the DataGrid control.

| Property | Type | Use | Description |
|---------------------------|--------|----------|--|
| <code>cellRenderer</code> | Object | Property | Specifies the class object reference or symbol linkage ID of the cell renderer to use. For more information, see Chapter 15, “Customizing Data Provider Controls,” on page 379 . |
| <code>columnCount</code> | Number | Property | Read-only property containing the number of columns. |
| <code>columnNames</code> | Array | Property | Specifies an Array containing the column names. You can specify the Array in MXML, as the following example shows: <code>columnNames="[foo, bar, baz]"</code> |
| <code>columns</code> | Array | Property | Specifies the set of field names within each data provider object displayed as column names. For more information, see “DataGridColumn syntax” on page 134 . |

| Property | Type | Use | Description |
|-------------------------------|----------|----------|---|
| <code>editable</code> | Boolean | Property | Specifies that the text area of the control is editable, <code>true</code> , or not, <code>false</code> . The default value is <code>false</code> . |
| <code>focusedCell</code> | Object | Property | Specifies the cell currently in focus based on its <code>columnIndex</code> and <code>itemIndex</code> . Origin is (0,0). For example: <code>focusedCell= "{columnIndex:2, itemIndex:3}"</code> |
| <code>headerHeight</code> | Integer | Property | Specifies the height of the header bar of the grid, in pixels. The default value is 20. |
| <code>iconField</code> | String | Property | While inherited from the List control, this property is not supported by the DataGrid control. |
| <code>iconFunction</code> | Function | Property | While inherited from the List control, this property is not supported by the DataGrid control. |
| <code>labelFunction</code> | Function | Property | Specifies a function used to decide the content to display for each cell in the DataGrid control. Flex calls this function once for each cell. This function takes a single argument, <code>item</code> , which is the item being rendered, and must return a string representing the text to display. For an example, see “Using a label function” on page 107 . |
| <code>resizableColumns</code> | Boolean | Property | Specifies whether the user can stretch the columns of the DataGrid control, <code>true</code> (default), or not, <code>false</code> . The value must be <code>true</code> for individual columns to be resizable. |
| <code>showHeaders</code> | Boolean | Property | Specifies whether the DataGrid control shows column headers, <code>true</code> (default), or not, <code>false</code> . |
| <code>sortableColumns</code> | Boolean | Property | Specifies whether the user can sort the columns of the DataGrid control by clicking the headers, <code>true</code> (default), or not, <code>false</code> . You must set the value to <code>true</code> for individual columns to be sortable, and to receive the <code>headerPress</code> event. |
| <code>cellEdit</code> | | Event | <ul style="list-style-type: none"> Specifies a handler for <code>cellEdit</code> events, which are broadcast when a cell value changes. |
| <code>cellFocusIn</code> | | Event | <ul style="list-style-type: none"> Specifies a handler for <code>cellFocusIn</code> events, which are broadcast when a particular cell gains focus. |
| <code>cellFocusOut</code> | | Event | <ul style="list-style-type: none"> Specifies a handler for <code>cellFocusOut</code> events, which are broadcast when a particular cell loses focus. |
| <code>cellPress</code> | | Event | <ul style="list-style-type: none"> Specifies a handler for <code>cellPress</code> events, which are broadcast when the user selects a cell. |
| <code>change</code> | | Event | Specifies a handler for <code>change</code> events. |
| <code>columnStretch</code> | | Event | <ul style="list-style-type: none"> Specifies a handler for <code>columnStretch</code> events, which are broadcast when a user stretches a column horizontally. |
| <code>headerRelease</code> | | Event | <ul style="list-style-type: none"> Specifies a handler for <code>headerRelease</code> events, which are broadcast when a column header is released. |

DataGridColumn syntax

You use the `<mx:DataGridColumn>` tag to configure a column of a DataGrid control. For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*. You specify the `<mx:DataGridColumn>` tag as a child of the `columns` property in MXML, in the following form:

```
<mx:DataGrid >
  <mx:columns>
    <mx:Array>
      <mx:DataGridColumn ... />
      <mx:DataGridColumn ... />
      ...
    </mx:Array>
  </mx:columns>
</mx:DataGrid>
```

The `<mx:DataGridColumn>` tag defines the properties in the following table:

| Property | Type | Use | Description |
|------------------------------|----------|----------|--|
| <code>columnName</code> | String | Property | (Required) The name of the data field associated with the column. |
| <code>cellRenderer</code> | | | Specifies the class object reference or symbol linkage ID of the <code>cellRenderer</code> to use. Assigns the <code>cellRenderer</code> to use for each cell of the column. |
| <code>dataTipField</code> | String | Property | Specifies the name of the field in the data provider to use as the data tip field for cells in the column. |
| <code>dataTipFunction</code> | Function | Property | Specifies a function that determines the data tip string to display for cells in the column. The callback function takes a single argument, which contains the data for the entire row, and returns a String. The signature of the callback function is: <i>mydataTipFunc(item:Object):String</i> |
| <code>editable</code> | Boolean | Property | Specifies whether a column is editable, <code>true</code> , or not, <code>false</code> . The default value is <code>false</code> . |
| <code>headerRenderer</code> | String | Property | Specifies the name of a class to be used to display the header of the column. |
| <code>headerText</code> | String | Property | Specifies the text for the header of this column. By default, the DataGrid control uses the name of the data field associated with the column as the column name. |
| <code>labelFunction</code> | Function | Property | Specifies a function that determines the string to display in the DataGrid control. The callback function takes a single argument containing the data for the entire row, and returns a String. The signature of the callback function is: <i>myLabelFunc(item:Object):String</i> |
| <code>resizable</code> | Boolean | Property | Specifies whether a column is resizable, <code>true</code> , or not, <code>false</code> . The default value is <code>true</code> . |

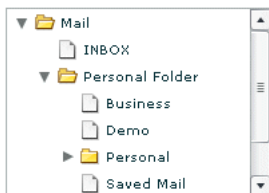
| Property | Type | Use | Description |
|---------------------|---------|----------|---|
| showDataTips | Boolean | Property | Specifies whether a column of cells shows DataTips, <i>true</i> , or not, <i>false</i> . The default value is <i>false</i> . DataTips are like ToolTips, but are for individual cells in DataGridColumns and Lists. |
| sortable | Boolean | Property | Specifies whether a column is sortable, <i>true</i> , or not, <i>false</i> . The default value is <i>true</i> . |
| sortOnHeaderRelease | Boolean | Property | Specifies whether a column is sorted, <i>true</i> , or not, <i>false</i> , when a user presses a column header. The default value is <i>true</i> . You can set this property to <i>true</i> only if <i>sortable</i> is <i>true</i> . |
| variableRowHeight | Boolean | Property | Specifies whether the height of rows is variable. The default value is <i>false</i> . If <i>true</i> , the <i>rowHeight</i> property is ignored and the <i>rowCount</i> property is read-only. |
| width | Number | Property | Specifies the width of a column, in pixels. The default value is 100 pixels. |
| wordWrap | Boolean | Property | Specifies whether the cells in a column wrap words to the next line, <i>true</i> , or not, <i>false</i> . The default value is <i>false</i> . |

Tree control

The Tree control lets a user view hierarchical data arranged as an expandable tree. Each item in a tree is called a *node* and can be either a leaf or a branch. A *branch* node can contain either leaf or branch nodes. A *leaf* node is an end point in the tree.

By default, a leaf is represented by a text label beside a file icon and a branch is represented by a text label beside a folder icon with a disclosure triangle that a user can open to expose children.

The following figure shows a Tree control:



The Tree control has the following default properties:

| Property | Default |
|----------------|---|
| preferred size | 200 pixels wide, and seven rows high, where each row is 22 pixels in height |
| minimum size | Width of icon plus 16 characters, and a height of three rows |
| maximum size | Undefined |

Creating a Tree control

You define a Tree control in MXML using the `<mx:Tree>` tag. The Tree control is derived from the List control and takes all of the properties and methods of the List control. For more information about using the List control, see [“List control” on page 105](#). Specify an `id` value if you intend to refer to a control elsewhere in your MXML, either in another tag or in an ActionScript block.

The Tree control uses a hierarchical data provider. For more information, see [“Using data providers with hierarchical controls” on page 101](#).

A Tree control gets its data from a data provider. Often, the data for a Tree control is retrieved from a server in the form of XML, but it can also be well-formed XML that you defined within the `<mx:Tree>` tag. The following code contains a single Tree control that defines the tree shown in the figure in [“Tree control” on page 135](#):

```
<mx:Tree id="tree1" >
  <mx:dataProvider>
    <mx:XML>
      <node label="Mail">
        <node label="INBOX"/>
        <node label="Personal Folder">
          <node label="Business" />
          <node label="Demo" />
          <node label="Personal" isBranch="true" />
          <node label="Saved Mail" />
        </node>
        <node label="Sent" />
        <node label="Trash"/>
      </node>
    </mx:XML>
  </mx:dataProvider>
</mx:Tree>
```

Node tags in the XML data can have any name. In this example, each node is named with the generic `<node>` tag. The Tree control reads the XML and builds the display hierarchy based on the nested relationship of the nodes. For the list of options to the individual nodes, see [“Node syntax” on page 141](#).

Nodes at the highest level are called *root* nodes and have no parent. A Tree control can have multiple root nodes. In this example, there is only one root node in the tree: “Mail”. However, if you add sibling nodes at that level in the XML, multiple root nodes appear in the Tree control.

A branch node can contain multiple child nodes, and appear as a folder icon with a disclosure triangle that lets users open and close the folder. Leaf nodes appear as a file icon and cannot contain child nodes.

When a Tree control displays a node, by default, it displays the value of the `label` property of the node as the text label.

Handling Tree control events

You typically use events to respond to user interaction with a Tree control. Since the Tree control is derived from the List control, you can use all of the events defined for the List control plus two events added by the Tree control: `nodeOpen` and `nodeClose`. The following example defines event handlers for the `change` and `nodeOpen` events:

```
<mx:Script>
  <![CDATA[
    function changeEvt(event) {
      forChange.text=event.target.selectedItem.attributes.label + " " +
        event.target.selectedItem.attributes.data;
    }

    function nodeOpenEvt(event) {
      forOpen.text=event.node.attributes.label + " " +
        event.node.attributes.data;
    }
  ]]>
</mx:Script>

<mx:Tree id="tree1" width="300" height="500"
  change="changeEvt(event)" nodeOpen="nodeOpenEvt(event)" >
  <mx:dataProvider>
    <mx:XML>
      <node label="Mail">
        <node label="INBOX"/>
        <node label="Personal Folder">
          <node label="Business" data="2"/>
          <node label="Demo" />
          <node label="Personal" isBranch="true" />
          <node label="Saved Mail" />
        </node>
        <node label="Sent" />
        <node label="Trash"/>
      </node>
    </mx:XML>
  </mx:dataProvider>
</mx:Tree>
<mx:TextArea id="forChange" width="150" />
<mx:TextArea id="forOpen" width="150" />
```

In this example, you define event handlers for the `change` and `nodeOpen` events. The Tree control broadcasts the `change` event when the user selects a tree item, and broadcasts the `nodeOpen` event when a user opens a branch node. For each event, the event handler displays the label and the data property, if any, in a TextArea control. In this example, only the *Business* node defines a data value; for all other nodes, the data value is undefined.

Expanding a tree node

By default, the Tree control displays the root nodes of the tree when it first opens. If you want to expand a node of the tree when the tree opens, you can use the `setIsOpen()` method of the Tree control. The following example calls this method as part of the handler for the `initialize` event to expand the first root node of the tree:

```
<mx:Script>
    <![CDATA[
        function initTree(){
            tree1.setIsOpen(tree1.getTreeNodeAt(0), true);
        }
    ]]>
</mx:Script>

<mx:Tree id="tree1" ... initialize="initTree()" >
    ...
</mx:Tree>
```

Editing a node label at runtime

You can use the `editable` property of the Tree control to make node labels editable at runtime. To edit a node label, the user selects the label, and then enters a new label or edits the existing label text. By default, node labels are not editable. Set the `editable` property to `true` to enable editing.

Changing Tree control icons

You can use the `folderOpenIcon`, `folderClosedIcon`, and `defaultLeafIcon` properties to control the Tree control icons. For example, the following code specifies a default icon, and icons for the open and closed states of branch nodes:

```
<mx:Tree folderOpenIcon="@Embed('open.jpg')"
    folderClosedIcon="@Embed('closed.jpg')"
    defaultLeafIcon="@Embed('def.jpg')">
```

If you want to remove the branch and leaf node icons so that the Tree control uses only a small triangle to signify branch nodes, you use the following code:

```
<mx:Tree folderOpenIcon="UIObject"
    folderClosedIcon="UIObject"
    defaultLeafIcon="UIObject" >
```

You can specify an icon displayed with each Tree item when you populate it using XML, as the following example shows:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

    <mx:Script>
        <![CDATA[
            [Embed(source="mm.jpg")]
            var iconSymbol1:String;
            [Embed(source="atom.jpg")]
            var iconSymbol2:String;
```

```

]]>
</mx:Script>

<mx:Tree iconField="icon">
  <mx:dataProvider>
    <mx:XML>
      <node label="New" icon="iconSymbol1">
        <node label="HTML Document" icon="iconSymbol1"/>
        <node label="Text Document" icon="iconSymbol2"/>
      </node>
      <node label="Close" icon="iconSymbol2"/>
    </mx:XML>
  </mx:dataProvider>
</mx:Tree>
</mx:Application>

```

In this example, you use the `iconField` property to specify the field of each item containing the icon. You use the `Embed` metadata to import the icons, then reference them in the XML definition.

Using the keyboard to edit labels

You can use the following keys to edit labels:

| Key | Description |
|---------------------|--|
| Enter | The selected row changes state from selected to editing, or from editing to selected. |
| ^. (Control+Period) | Cancels the edit, restores the text, and changes the row state from editing to selected. |
| TAB | When in editing mode, accepts the current changes, selects the row below, and goes into editing mode. If at the last element in the tree or not in editing mode, sends focus to the next control, as usual. |
| Shift-TAB | When in editing mode, accepts the current changes, selects the row above, and goes into editing mode. If at the first element in the tree or not in editing mode, sends focus to the previous control, as usual. |

Editing events

To support label editing, the `Tree` control uses the following events:

`cellEdit` Broadcast when a tree label value changes.

`cellFocusIn` Broadcast when a user selects a label hit area or tabs to it, and the label receives focus.

`cellFocusOut` Broadcast when a label loses focus.

`cellPress` Broadcast when a user selects a label with the mouse pointer.

User interaction

When a Tree control has focus from clicking or tabbing, you use the following keys to control it:

| Key | Description |
|-------------|---|
| Down arrow | Moves the selection down one. |
| Up arrow | Moves the selection up one. |
| Right arrow | Opens a selected branch node. If a branch is already open, moves to the first child node. |
| Left arrow | Closes a selected branch node. If a leaf node of a closed branch node is currently selected, selects the parent node. |
| Space | Opens or closes a selected branch node. |
| End | Moves the selection to the bottom of the list. |
| Home | Moves the selection to the top of the list. |
| Page down | Moves the selection down one page. |
| Page up | Moves the selection up one page. |
| Control | Allows multiple noncontiguous selections. |
| Shift | Allows multiple contiguous selections. |

Tree control syntax

You use the `<mx:Tree>` tag to define a Tree control. For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*. The Tree control is derived from the List control. It takes all of the properties and methods of the List control. The Tree control also defines the optional properties described in the following table:

| Property | Type | Use | Description |
|-------------------------------|-------------|----------|---|
| <code>dataProvider</code> | Object | Property | Specifies a hierarchical data structure to populate the Tree control. You typically use an <code><mx:Model></code> or <code><mx:XML></code> tag to define this data structure. For more information, see “Using data providers with hierarchical controls” on page 101 . |
| <code>editable</code> | Boolean | Property | Specifies that node labels in the Tree control can be edited at runtime, <code>true</code> , or not, <code>false</code> . The default value is <code>false</code> . |
| <code>firstVisibleNode</code> | Node object | Property | Specifies the first node at the top of the Tree. Setting this property is analogous to setting the <code>vPosition</code> of the List control. The default is the first node in the Tree. If the node isn’t currently viewable, for example because it is under a nonexpanded node, setting this property has no effect. |
| <code>selectedNode</code> | Node object | Property | Specifies the selected node in the tree. You can specify a single node, or an Array of nodes. |
| <code>cellEdit</code> | | Event | <ul style="list-style-type: none">• Broadcast when a tree label value changes. |

| Property | Type | Use | Description |
|---------------------------|------|-------|--|
| <code>cellFocusIn</code> | | Event | <ul style="list-style-type: none"> Broadcast when a user selects a label hit area or tabs to it and the label receives focus. |
| <code>cellFocusOut</code> | | Event | <ul style="list-style-type: none"> Broadcast when a label loses focus. |
| <code>cellPress</code> | | Event | <ul style="list-style-type: none"> Broadcast when a user selects a label with the mouse pointer. |
| <code>nodeOpen</code> | | Event | Specifies a handler for branch <code>nodeOpen</code> events. |
| <code>nodeClose</code> | | Event | Specifies a handler for branch node close events. |

Node syntax

For each node in the tree, you can define the following properties:

| Property | Type | Description |
|-----------------------|--------|---|
| <code>data</code> | Any | Specifies the data to associate with a node, if any. |
| <code>label</code> | String | Specifies the text to be displayed in the tree for a node. |
| <code>isBranch</code> | String | Specifies whether the node is a branch, <code>true</code> , or not, <code>false</code> . If not specified, the Tree control determines whether the node is a branch based on whether it contains any child nodes. |

ComboBox control

The ComboBox control is a drop-down list from which the user can select a single value. Its functionality is very similar to that of the SELECT form element in HTML.

The following figure shows a ComboBox control:



In its editable state, the user can type text directly into the top of the list, or select one of the preset values from the list. In its noneditable state, as the user types a letter, the drop-down list opens and scrolls to the value that most closely matches the one being entered; matching is only performed on the first letter that the user types.

If the drop-down list hits the lower boundary of the application, it opens upward. If a list item is too long to fit in the horizontal display area, it is truncated to fit. If there are too many items to display in the drop-down list, a vertical scroll bar appears.

The ComboBox control has the following default properties:

| Property | Default |
|----------------|--|
| preferred size | 100 pixels wide. When the drop-down list is not visible, the default height is based on the text size. The default drop-down list height is five rows, or the number of entries in the drop-down list, whichever is smaller. The default height of each entry in the drop-down list is 22 pixels. |
| minimum size | 0 |
| maximum size | No limit. |
| dropdownWidth | The width of the ComboBox control. |
| rowCount | 5 |

Creating a ComboBox control

You use the `<mx:ComboBox>` tag to define a ComboBox control in MXML. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The ComboBox control uses a list-based data provider. For more information, see [“Using data providers with list-based components” on page 96](#).

You specify the data for the ComboBox control using the `<mx:dataProvider>` child tag of the `<mx:ComboBox>` tag. The `<mx:dataProvider>` tag lets you specify data in several different ways. In the simplest case for creating a ComboBox control, you use the `<mx:dataProvider>`, `<mx:Array>`, and `<mx:String>` tags to define the entries as an Array of Strings, as the following example shows:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:ComboBox>
    <mx:dataProvider>
      <mx:Array>
        <mx:String>AL</mx:String>
        <mx:String>AK</mx:String>
        <mx:String>AR</mx:String>
      </mx:Array>
    </mx:dataProvider>
  </mx:ComboBox>
</mx:Application>
```

The index of items in the ComboBox control is zero-based, which means that values are 0, 1, 2, ..., $n - 1$, where n is the total number of items. The value of the item is its label text.

You typically use events to handle user interaction with a ComboBox control. The following example adds a handler for a change event and an open event to the ComboBox control. Flex broadcasts a change event when the value of the control changes due to user interaction, and broadcasts the open event when the ComboBox control opens.

```
<mx:Script>
  <![CDATA[
```

```

        function openEvt(event) {
            forChange.text="opened";
        }
        function changeEvt(event) {
            forChange.text=event.target.selectedItem + " " +
                event.target.selectedIndex;
        }
    ]]]>
</mx:Script>

<mx:ComboBox open="openEvt(event)" change="changeEvt(event)" >
    ...
</mx:ComboBox>
<mx:TextArea id="forChange" width="150" />

```

The `target` property of the object passed to the event handler contains a reference to the `ComboBox` control, and the `target.selectedItem` field contains a reference to the selected item. If you populate the `ComboBox` control with an Array of Strings, the `target.selectedItem` field contains a string. If you populate it with an Array of Objects, the `target.selectedItem` field contains a reference to the object that correspond to the selected item.

In this example, you use two properties of the `ComboBox` control, `selectedItem` and `selectedIndex`, in the event handlers. Every `change` event updates the `TextArea` control with the label of the selected item and the item's index in the control, and an `open` event clears the current text in the `TextArea` control.

Using Objects to populate a ComboBox control

You can populate a `ComboBox` with an Array of Objects. Objects let you define a `label` property that contains the string displayed in the `ComboBox` control, and a `data` property that contains any data that you want to associate with the label, as the following example shows:

```

<mx:ComboBox>
    <mx:dataProvider>
        <mx:Array>
            <mx:Object label="AL" data="Montgomery"/>
            <mx:Object label="AK" data="Juneau"/>
            <mx:Object label="AR" data="Little Rock"/>
        </mx:Array>
    </mx:dataProvider>
</mx:ComboBox>

```

The `label` property contains the state name and the `data` property contains the name of its capital. The following example shows a handler for the `change` event for this `ComboBox` control:

```

<mx:Script>
    <![CDATA[
        function openEvt(event) {
            forChange.text="";
        }
        function changeEvt(event) {
            forChange.text=event.target.selectedItem.data + " " +
                event.target.selectedIndex;
        }
    ]]]>

```

```

    }
  ]]]>
</mx:Script>

<mx:ComboBox open="openEvt(event)" change="changeEvt(event)" >
  ...
</mx:ComboBox>
<mx:TextArea id="forChange" width="150" />

```

In this example, the `selectedIndex` property contains the index of the item selected in the `ComboBox` control, and the `selectedItem` property contains a copy of the object defining the selected item. You use `selectedItem.data` to access the `data` property, and `selectedItem.label` to access the `label` property. Every `change` event updates the `TextArea` control with the `data` property of the selected item and the item's index in the control, and an `open` event clears the current text in the `TextArea` control.

By default, the `ComboBox` control expects each object to contain a property named `label` that defines the text that appears in the `ComboBox` control for the item. If each Object does not contain a `label` property, you can use the `labelField` property of the `ComboBox` control to specify the property name, as the following example shows:

```

<mx:ComboBox labelField="state" open="openEvt(event)"
  change="changeEvt(event)">
  <mx:dataProvider>
    <mx:Array>
      <mx:Object state="AL" capital="Montgomery"/>
      <mx:Object state="AK" capital="Juneau"/>
      <mx:Object state="AR" capital="Little Rock"/>
    </mx:Array>
  </mx:dataProvider>
</mx:ComboBox>

```

You also have to rewrite the handler for the `change` event since the object contains a property named `capital` with the name of the state capital, instead of `data`, as the following example shows:

```

function changeEvt(event) {
  forChange.text=event.target.selectedItem.capital + " " +
    event.target.selectedIndex;
}

```

Broadcasting a change event

The `ComboBox` control broadcasts a `change` event for the following user actions:

- If the user closes the drop-down list using a mouse click, Enter key, or Control+Up key, and the selected item is different from the previously selected item.
- If the drop-down list is currently closed, and the user presses the Up, Down, Page Up, or Page Down key to select a new item.
- If the `ComboBox` control is editable, and the user types into the control, Flex broadcasts a single `change` event each time the text field of the control changes.

User interaction

A ComboBox control can be noneditable or editable. In a noneditable ComboBox control, a user can make a single selection from a drop-down list. In an editable ComboBox control, a user can enter text directly into a text field at the top of the list, and can also select an item from the drop-down list.

When the user makes a selection in the ComboBox control list, the label of the selection is copied to the text field at the top of the ComboBox control.

When a ComboBox control has focus and is editable, all keystrokes go to the text box and are handled according to the rules of the TextInput control (see [“TextInput control” on page 88](#)), with the exception of the following keys:

| Key | Description |
|--------------|---|
| Control+Down | Opens the drop-down list and gives it focus. |
| Shift +Tab | Moves focus to the previous object in the list. |
| Tab | Moves focus to the next object in the list. |

When a ComboBox control has focus and is noneditable, alphanumeric keystrokes move the selection up and down the drop-down list to the next item with the same first character. You can also use the following keys to control a noneditable ComboBox control:

| Key | Description |
|--------------|--|
| Control+Down | Opens the drop-down list and gives it focus. |
| Control+Up | Closes the drop-down list, if open. |
| Down | Moves the selection down one item. |
| End | Moves the selection to the bottom of the list. |
| Escape | Closes the drop-down list and returns focus to the ComboBox control. |
| Enter | Closes the drop-down list and returns focus to the ComboBox control. |
| Home | Moves the selection to the top of the list. |
| Page down | Fills the ComboBox control with the next set of undisplayed items. |
| Page up | Fills the ComboBox control with the previous set of undisplayed items. |

When the drop-down list of a ComboBox control has focus, alphanumeric keystrokes move the selection up and down the drop-down list to the next item with the same first character. You can also use the following keys to control a drop-down list:

| Key | Description |
|------------|--|
| Control+Up | If the drop-down list is open, returns focus to the ComboBox control's text field and the drop-down list closes. |
| Down | Moves the selection down one item. |
| End | Moves the insertion point to the end of the text field. |

| Key | Description |
|------------|---|
| Enter | If the drop-down list is open, returns focus to the text field and the drop-down list closes. |
| Escape | If the drop-down list is open, returns focus to the text field and the drop-down list closes. |
| Home | Moves the insertion point to the beginning of the text field. |
| Page Down | Displays the next set of undisplayed items. |
| Page Up | Displays the previous set of undisplayed items. |
| Tab | Moves the focus to the next object in the list. |
| Shift+End | Selects the text from the insertion point to the End position. |
| Shift+Home | Selects the text from the insertion point to the Home position. |
| Shift+Tab | Moves the focus to the previous object. |
| Up | Moves the selection up one item. |

Note: The page size used by the Page Up and Page Down keys is one less than the number of items that fit in the display. For example, paging down through a 10-line drop-down list shows items 0-9, 9-18, 18-27, and so on, with one item overlapping per page.

ComboBox control syntax

You use the `<mx:ComboBox>` tag to define a ComboBox control. For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*. The ComboBox control defines the optional properties described in the following table:

| Property | Type | Use | Description |
|----------------------------|----------|----------|---|
| <code>dataProvider</code> | Array | Property | Specifies the data used to populate the ComboBox control. |
| <code>dropdownWidth</code> | Number | Property | Specifies the width of the ComboBox control, in pixels. The default value is a size wide enough to hold the text. |
| <code>editable</code> | Boolean | Property | Specifies that the text area of the ComboBox control is editable, <code>true</code> , or not, <code>false</code> . The default value is <code>false</code> . An editable ComboBox control can have values entered into the text field that do not show up in the ComboBox list. |
| <code>labelField</code> | String | Property | Specifies the name of the field in the objects of the <code>dataProvider</code> array to use as the label field. If omitted, the data provider must contain a field named <code>label</code> , or the <code>dataProvider</code> property must contain an Array of Strings. |
| <code>labelFunction</code> | Function | Property | Specifies a function to determine the label from information in each data provider object. For an example, see “Using a label function” on page 107 . |
| <code>length</code> | Number | Property | A read-only property that contains the number of items in the ComboBox control. |

| Property | Type | Use | Description |
|----------------------------|--|----------|--|
| <code>restrict</code> | String | Property | Specifies the set of characters that a user can enter into the text field. For more information, see the <code>restrict</code> property of the <code>TextArea</code> control, “TextArea control syntax” on page 86 . |
| <code>rowCount</code> | Number | Property | Specifies the maximum number of rows visible in the <code>ComboBox</code> control list. The default value is 5. |
| <code>selectedIndex</code> | Number | Property | <p>Contains the index of the selected item in the drop-down box.</p> <p>Setting this property sets the current index, and displays the associated label in the text field. The default value is <code>undefined</code>.</p> <p>If the control is editable, users can also set this property by editing the text field of the <code>ComboBox</code> control.</p> <p>Typing into the text field of an editable <code>ComboBox</code> sets the selected index to <code>undefined</code>.</p> <p>If the selected index is out of range, the assignment is ignored.</p> |
| <code>selectedItem</code> | Based on type of <code>dataProvider</code> | Property | <p>Contains a reference to the selected item in the data provider. Modifying this value modifies the data provider.</p> <p>If the <code>ComboBox</code> control is editable, <code>selectedItem</code> contains <code>undefined</code> if the user types any text in the text field. It will only have a value if the user selects an item from the drop-down list, or if it's set programmatically.</p> |
| <code>text</code> | String | Property | For an editable <code>ComboBox</code> control, sets the text displayed in the text area. For a noneditable control, it does nothing. |
| <code>value</code> | String | Property | <p>If the control is editable, this read-only property contains the text of the text field portion of the control, including text entered directly into the control by the user.</p> <p>If the control is noneditable, it contains the value of the selected item. The value is either the data field of an object specified by <code>dataProvider</code>, or the <code>label</code> field.</p> |
| <code>change</code> | | Event | Specifies a handler for <code>change</code> events, which are broadcast when the value of the control changes as a result of user interaction. |
| <code>close</code> | | Event | Specifies a handler for <code>close</code> events, which are broadcast when a <code>ComboBox</code> control begins to retract. |
| <code>enter</code> | | Event | Specifies a handler for <code>enter</code> events, which are broadcast when the user presses the Enter key in an editable control. |
| <code>itemRollOver</code> | | Event | Specifies a handler for <code>itemRollOver</code> events, which are broadcast when <code>ComboBox</code> control list items are rolled over. |
| <code>itemRollOut</code> | | Event | Specifies a handler for <code>itemRollOut</code> events, which are broadcast when <code>ComboBox</code> control list items are rolled out. |

| Property | Type | Use | Description |
|---------------------|------|-------|---|
| <code>open</code> | | Event | Specifies a handler for <code>open</code> events, which are broadcast when a ComboBox control begins to expand. |
| <code>scroll</code> | | Event | Specifies a handler for <code>scroll</code> events, which are broadcast when the ComboBox control list is scrolled. |

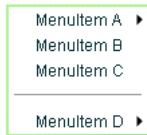
Menu control

You use the Menu control to create a pop-up menu of individually selectable choices. You use ActionScript to pop up a Menu control in response to a user action, typically as part of an event handler. Since you create a Menu control in response to an event, it does not have an MXML tag; you can only create one in ActionScript.

After a Menu opens, it remains visible until it is closed by a script, or the user selects another component in the application, or the user selects an enabled menu item.

If you want a static menu, meaning one that stays visible all the time, use the MenuBar control. The MenuBar control creates a horizontal menu bar with menus that extend under each menu bar item. For more information on the MenuBar control, see [“MenuBar control” on page 153](#).

The following figure shows a Menu control:



In this example, items MenuItem A and MenuItem D open submenus.

The Menu control has the following default properties:

| Property | Default |
|-----------------------------|--|
| <code>preferred size</code> | The width is determined from the Menu text. The default height is the number of menu rows multiplied by 22 pixels per row. |

Creating a Menu control

You create a Menu control in ActionScript, not in MXML, using the methods `Menu.createMenu()` and `Menu.show()`. The `createMenu()` method creates an instance of a Menu control. You pass to this method the data provider for the Menu control. The `show()` method makes the Menu control visible.

The Menu control uses a hierarchical data provider. For more information, see [“Using data providers with hierarchical controls” on page 101](#).

In this example, you use the `<mx:XML>` tag to define the data for the Menu control and a Button control to trigger the event that opens the Menu control:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
```

```

<mx:Script>
  <![CDATA[
    // Import the Menu control.
    import mx.controls.Menu;
    // Define the Menu control.
    function createAndShow(){
      var myMenu:Menu = Menu.createMenu(null, myMenuData);
      myMenu.show(10, 10);
    }
  ]]>
</mx:Script>

<!-- Define the menu data. -->
<mx:XML id="myMenuData">
  <menuItem label="MenuItem A" >
    <menuItem label="SubMenuItem 1-A" />
    <menuItem label="SubMenuItem 2-A" />
  </menuItem>
  <menuItem label="MenuItem B" />
  <menuItem label="MenuItem C" type="check" />
  <menuItem type="separator" />
  <menuItem label="MenuItem D" >
    <menuItem label="SubMenuItem 1-D" type="radio" groupName="one" />
    <menuItem label="SubMenuItem 2-D" type="radio" groupName="one" />
    <menuItem label="SubMenuItem 3-D" type="radio" groupName="one" />
  </menuItem>
</mx:XML>

<mx:VBox>
  <!-- Define a Button control to open the menu -->
  <mx:Button id="myButton" label="Open Popup" click="createAndShow()" />
</mx:VBox>
</mx:Application>

```

Node tags in the XML data can have any name. In the preceding example, each node is named with the generic `<menuItem>` tag, but you could use `<node>`, `<myNode>`, and so on. The Menu control reads through the XML and builds the display hierarchy based on the nested relationship of the nodes. For more information, see [“MenuItem syntax” on page 158](#).

The default location of the Menu control is the top, left corner of your application. You can pass `x` and `y` arguments to the `show()` method to control position.

You use the `<menuItem>` tag to define the items of the Menu control, both the top-level menu items and any submenu items. You use the `type` property of the `<menuItem>` tag to specify the type of the menu item as one of the following:

normal Selecting these items triggers a change event or opens a submenu, if the item has child `<menuItem>` tags. This is the default.

check Selecting these items toggles the menu item’s `selected` property between `true` and `false` values. When the menu item is in the `true` state, it displays a check mark in the menu next to the item’s label.

radio These items operate in groups, much like `RadioButton` controls; you can select only one radio menu item in each group at a time. The example in this section defines three submenu items as radio buttons within the group “one”.

When selected, the radio item’s `selected` property is set to `true`, and the `selected` property of all other radio items in the group is set to `false`. The `Menu` control displays a solid circle next to the radio button that is currently selected. The `selection` property of the radio group is set to the label of the selected menu item.

separator These items provide a simple horizontal line that divides the items in a menu into different visual groups.

Handling Menu control events

User interaction with a `Menu` control is event-driven. Besides the events that it inherits from the `UIObject` and `UIComponent` classes, the `Menu` control defines the following additional event types:

change Broadcast when a user selects an enabled menu item of type `normal`, `check`, or `radio`. This event is not broadcast when a user selects a menu item of type `separator`, a menu item that opens a submenu, or a disabled menu item.

menuHide Broadcast when the entire menu or a submenu closes.

menuShow Broadcast when the entire menu or a submenu opens.

rollOut Broadcast when the mouse pointer rolls off of a `Menu` item.

rollOver Broadcast when the mouse pointer rolls onto a `Menu` item.

The event object passed to the event handler might contain one or all of the following properties:

menuBar The `MenuBar` control instance that is the parent of the selected `Menu` control, or `undefined` when the target `Menu` control does not belong to a `MenuBar`. The data type is `MenuBar`. For more information, see [“MenuBar control” on page 153](#).

menu A reference to the `Menu` control of the selected item, of type `Menu`.

menuItem The selected menu item. Access the menu item properties as follows:

```
eventobj.menuItem.attributes.attributeName
```

For example:

```
var itemLabel:String = event.menuItem.attributes.label;  
var itemSelectedValue:Number=event.menuItem.attributes.selected;
```

For more information on menu item properties, see [“MenuItem syntax” on page 158](#).

For a complete description of the event object for each event, see [“Menu control syntax” on page 152](#).

The following example creates an event handler for the `Menu` control:

```
<?xml version="1.0"?>  
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">  
  
    <mx:Script>
```

```

<![CDATA[

    // Import the Menu control and Delegate.
    import mx.controls.Menu;
    import mx.utils.Delegate;
    //Define a variable for the Menu control.
    var myMenu:Menu;

    // Define the event handler that creates the menu,
    // and adds event listeners.
    function createAndShow() {
        myMenu = Menu.createMenu(null, myMenuData);
        myMenu.addEventListener("change",
            Delegate.create(this, menuShowInfo));
        myMenu.addEventListener("menuHide",
            Delegate.create(this, menuShowInfo));
        myMenu.addEventListener("rollOver",
            Delegate.create(this, menuShowInfo));
        myMenu.show(10, 10);
    }

    // Define the event handler for the Menu events.
    function menuShowInfo(event) {
        taMenuShow.text=event.menuItem.attributes.label + " " +
            event.menuItem.attributes.selected;
    }
]]>
</mx:Script>

<!-- Define the menu data. -->
<mx:XML id="myMenuData">
    <menuitem label="MenuItem A" >
        <menuitem label="SubMenuItem 1-A" />
        <menuitem label="SubMenuItem 2-A" />
    </menuitem>
    <menuitem label="MenuItem B" />
    <menuitem label="MenuItem C" type="check" />
    <menuitem type="separator" />
    <menuitem label="MenuItem D" >
        <menuitem label="SubMenuItem 1-D" type="radio" groupName="one" />
        <menuitem label="SubMenuItem 2-D" type="radio" groupName="one" />
        <menuitem label="SubMenuItem 3-D" type="radio" groupName="one" />
    </menuitem>
</mx:XML>

<mx:VBox>
    <!-- Define a Button control to open the menu. -->
    <mx:Button id="myButton" label="Open Popup" click="createAndShow()" />
    <mx:TextArea id="taMenuShow" />
</mx:VBox>
</mx:Application>

```

In this example, the event handler writes a string to the `TextArea` control containing the value of the `label` and `selected` properties of the selected menu item. The value of the `selected` property is either `true` or `false` for a check or radio menu item, and undefined for all other menu item types. For more information on the properties of a menu item, see [“MenuItem syntax” on page 158](#).

You need to use the `Delegate` class to ensure that the event handler executes in the correct scope. For more information, see [Chapter 12, “Working with ActionScript in Flex,” on page 319](#).

User interaction

You can use the mouse or the keyboard to interact with a `Menu` control. Clicking selects a menu item and closes the menu, except with the following types of menu items:

Disabled items or separators Rollovers and clicks have no effect and the menu remains visible.

Submenu anchors Rollovers activate the submenu; clicks have no effect; rolling onto any menu item other than one of the submenu items closes the submenu.

When a `Menu` control has focus, you can use the following keys to control it:

| Key | Description |
|-------------|--|
| Down arrow | Moves the selection down and up the rows of the menu. The selection loops at the top or bottom row. |
| Up arrow | |
| Right arrow | Opens a submenu, or moves selection to the next menu in a menu bar. |
| Left arrow | Closes a submenu and returns focus to the parent menu (if a parent menu exists), or moves selection to the previous menu in a menu bar (if the menu bar exists). |
| Enter | Opens a submenu, or clicks and releases on a row if a submenu does not exist. |

Menu control syntax

You create a `Menu` control in ActionScript using the `createMenu()` method. For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*. The following table describes the properties defined by the `Menu` control:

| Property | Type | Use | Description |
|---------------------------|--------------------|----------|---|
| <code>dataProvider</code> | <code>Array</code> | Property | Specifies an <code>Array</code> of simple types or objects. |
| <code>change</code> | | Event | <ul style="list-style-type: none">Broadcast when a user selects an item in the <code>Menu</code> control. This event is also broadcast when the user presses the <code>Enter</code> or <code>Space</code> key when a leaf node is selected, but not when a branch node is selected. |
| <code>menuHide</code> | | Event | <ul style="list-style-type: none">Broadcast when a menu closes. |
| <code>menuShow</code> | | Event | <ul style="list-style-type: none">Broadcast when the entire menu or a submenu opens. |
| <code>rollOut</code> | | Event | Broadcast when the cursor rolls off of a <code>Menu</code> item. |
| <code>rollOver</code> | | Event | Broadcast when the cursor rolls over a <code>Menu</code> item. |

MenuBar control

A MenuBar control defines a horizontal menu bar containing one or more submenus. A MenuBar control supports the same syntax and events as the Menu control. Unlike the Menu control, a MenuBar control is static; that is, it does not function as a pop-up menu, but is always visible in your application.

For more information on the Menu control, see [“Menu control” on page 148](#).

The following figure shows a MenuBar control:



When a user selects a top-level menu item, the MenuBar control opens a submenu. The submenu stays open until the user selects another top-level menu item, the user selects a submenu item, or the user clicks outside the MenuBar area.

The MenuBar control has the following default properties:

| Property | Default |
|----------------|--|
| preferred size | The width is determined from the menu text. The default height is 22 pixels. |

Creating a MenuBar control

You define a MenuBar control in MXML using the `<mx:MenuBar>` tag. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The Menu control uses a hierarchical data provider. For more information, see [“Using data providers with hierarchical controls” on page 101](#).

You specify the data for the MenuBar control as XML formatted data using the `<mx:dataProvider>` child tag of the `<mx:MenuBar>` tag. The `<mx:dataProvider>` tag lets you specify data in several different ways. You can populate a MenuBar control from data retrieved from a server, or it can be well-formed XML defined within the `<mx:MenuBar>` tag.

In the simplest case for creating a MenuBar control, you use the `<mx:dataProvider>` and `<menuItem>` tags to define the entries as an Array of strings, as the following example shows:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

  <!-- Define the menu. -->
  <mx:MenuBar id="myMenuBar">
    <mx:dataProvider>
      <mx:XML>
        <menuItem label="MenuItem A" >
          <menuItem label="SubMenuItem 1-A" />
          <menuItem label="SubMenuItem 2-A" />
        </menuItem>
        <menuItem label="MenuItem B" />
        <menuItem label="MenuItem C" />
        <menuItem label="MenuItem D" >
```

```

        <menuitem label="SubMenuItem 1-D" type="radio" groupName="one" />
        <menuitem label="SubMenuItem 2-D" type="radio" groupName="one" />
        <menuitem label="SubMenuItem 3-D" type="radio" groupName="one" />
    </menuitem>
</mx:XML>
</mx:dataProvider>
</mx:MenuBar>
</mx:Application>

```

Node tags in the XML data can have any name. In the sample above, each node is named with the generic `<menuitem>` tag, but you can have used `<node>`, `<myNode>`, and so on. The Menu control reads through the XML and builds the display hierarchy based on the nested relationship of the nodes. For a list of the options that you can specify for each menu item, see [“Menuitem syntax” on page 158](#).

The top level nodes in the MenuBar control correspond to the labels that appear in the bar. Therefore, in this example, the MenuBar control displays the four labels shown in the preceding figure.

Importing XML data for the data provider

The example in the previous section defined the data provider for the MenuBar control within the `<mx:MenuBar>` tag. You can also import that XML from a file, as the following example shows:

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">

    <!-- Import the XML data. -->
    <mx:XML id="menuDS1" source="menuSrc.xml"/>
    <mx:MenuBar id="menu1" width="50%" dataProvider="{menuDS1}"/>

    <!-- Define the XML data inline. -->
    <mx:MenuBar id="menu2" width="50%">
        <mx:dataProvider>
            <mx:XML>
                <node label="File">
                    <node label="Load" />
                    <node label="Save" />
                    <node label="Expand" />
                    <node label="Exit" />
                </node>
            </mx:XML>
        </mx:dataProvider>
    </mx:MenuBar>
</mx:Application>

```

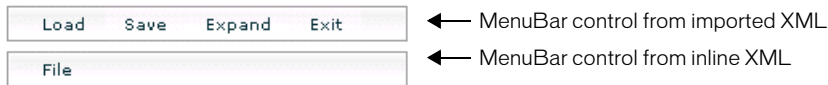
Where the XML defined in the menuSrc.xml file contains the following content:

```

<node label="File">
    <node label="Load" />
    <node label="Save" />
    <node label="Expand" />
    <node label="Exit" />
</node>

```

This code creates the two MenuBar controls shown in the following figure:



Flex works differently for imported XML than for XML defined within the `<mx:MenuBar>` tag. When you define the XML within the `<mx:MenuBar>` tag, Flex cannot assume that you created a valid root node, so Flex always creates one. Therefore, the node labeled `File` becomes the child node of the root node that Flex created.

When you load XML from an external file, Flex requires that you include a valid root node. Therefore, Flex considers the node labeled `File` in the imported XML file to be the root node and it does not appear in the MenuBar control.

If you want the MenuBar control defined using imported XML to look the same as the one defined XML defined within the `<mx:MenuBar>` tag, add a root node definition to the imported XML file, as the following example shows:

```
<node label="RootNode">
  <node label="File">
    <node label="Load" />
    <node label="Save" />
    <node label="Expand" />
    <node label="Exit" />
  </node>
</node>
```

Adding a menu item

The MenuBar control has an ActionScript API that you can use to manipulate it. For example, you can use the `MenuBar.addMenu()` method to add a menu item, as the following example shows:

```
<mx:TextInput id="input" text="type in new menu text" />
<mx:Button label="Add Menu" click="myMenuBar.addMenu(input.text)" />
```

You can modify this example to add a menu item to a submenu of the MenuBar control, as the following example shows:

```
<mx:Button label="Add Menu"
  click="myMenuBar.getMenuAt(0).addMenuItem(input.text)" />
```

This example uses the `getMenuAt()` method to access the first submenu of the MenuBar control, then uses `addMenuItem()` method to add the new menu item. The index of the submenu items starts at 0.

For additional methods of the MenuBar control, see *Flex ActionScript and MXML API Reference*.

Differences between the Menu and MenuBar controls

Although based on the Menu control, the MenuBar control differs in the following ways:

- The MenuBar control is a static control that always appears in your application. Therefore, you do not use the `show()` or `hide()` methods with the MenuBar control.
- The top-level nodes of a MenuBar control cannot have a type of `separator`.
- A change event is not broadcast when you select a top-level menu item. A change event is broadcast when you select an entry from a submenu.

Handling MenuBar control events

User interaction with a MenuBar control is event driven. Besides the events that it inherits from `UIObject` and `UIComponent`, the MenuBar control defines the additional event types:

`change` Broadcast when a user selects an enabled menu item of type `normal`, `check`, or `radio`. This event is not broadcast when a user selects a menu item of type `separator`, a menu item that opens a submenu, or a disabled menu item.

`menuHide` Broadcast when the entire menu or a submenu closes.

`menuShow` Broadcast when the entire menu or a submenu opens.

`rollOut` Broadcast when the mouse pointer rolls off of a MenuBar item.

`rollOver` Broadcast when the mouse pointer rolls onto a MenuBar item.

For a complete description of the event object for each event, see [“MenuBar control syntax” on page 157](#).

The following example creates an event handler for the `change` event:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

  <mx:Script>
    <![CDATA[
      function changeEvt(event) {
        forChange.text=event.menuItem.attributes.label + " " + " " +
          event.menuItem.attributes.selected;
      }

      function closeEvt() {
        forClose.text="menu closed";
      }

      function rollOverEvt(event) {
        forRollOver.text=event.menuItem.attributes.label + " " + " " +
          event.menuItem.attributes.selected;
      }
      function openEvt() {
        forClose.text="";
        forChange.text="";
      }
    ]]>
  </mx:Script>
</mx:Application>
```

```

</mx:Script>

<mx:MenuBar id="menuBar1" change="changeEvt(event)" menuHide="closeEvt()"
menuShow="openEvt()" rollover="rolloverEvt(event)" >
  <mx:dataProvider>
    <mx:XML>
      <menuItem label="MenuItem A" >
        <menuItem label="SubMenuItem 1-A" />
        <menuItem label="SubMenuItem 2-A" />
      </menuItem>
      <menuItem label="MenuItem B" />
      <menuItem label="MenuItem C" />
      <menuItem label="MenuItem D" >
        <menuItem label="SubMenuItem 1-D" type="radio" groupName="one" />
        <menuItem label="SubMenuItem 2-D" type="radio" groupName="one" />
        <menuItem label="SubMenuItem 3-D" type="radio" groupName="one" />
      </menuItem>
    </mx:XML>
  </mx:dataProvider>
</mx:MenuBar>

<mx:TextArea id="forChange" width="150" />
<mx:TextArea id="forClose" />
<mx:TextArea id="forRollOver" width="150" />
</mx:Application>

```

In this example, the event handler writes strings to the `TextArea` controls containing the value of the `label` and selected properties of the selected menu item. The value of the selected property is `true` or `false` for a check or radio menu item, and `undefined` for all other menu item types.

User interaction

The user interaction of the `MenuBar` control is the same as for the `Menu` control. For more information, see [“User interaction” on page 152](#).

MenuBar control syntax

You use the `<mx:MenuBar>` tag to define a `MenuBar` control. For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*. The following table describes the properties defined by the `MenuBar` control:

Note: The `MenuBar` control inherits the `borderStyle` property, so you cannot set it.

| Property | Type | Use | Description |
|---------------------------|--------|----------|--|
| <code>dataProvider</code> | Array | Property | Specifies an Array of simple types or Objects. |
| <code>labelField</code> | String | Property | Specifies the name of the field in the <code>dataProvider</code> array to use as the text for each menu item. If omitted, the data provider must contain a field named <code>label</code> , or the <code>dataProvider</code> property must contain an Array of Strings. |

| Property | Type | Use | Description |
|---------------|----------|----------|--|
| labelFunction | Function | Property | <p>Specifies a function that determines what displays as the text for each menu item.</p> <p>The function accepts the XML node associated with an item as a parameter, and returns a string used as label text. This property is propagated to any menus created from the MenuBar control.</p> <p>For an example, see “Using a label function” on page 107.</p> |
| menuHide | | Event | <ul style="list-style-type: none"> • Broadcast when a menu closes. |
| change | | Event | <ul style="list-style-type: none"> • Broadcast when a user selects an item in the MenuBar control. |
| menuShow | | Event | <ul style="list-style-type: none"> • Broadcast when the entire menu or a submenu opens. |
| rollOut | | Event | <p>Broadcast when the user moves the mouse pointer off of a MenuBar item. The event object contains the following properties:</p> <ul style="list-style-type: none"> • target Contains a reference to the MenuBar control. • type Contains the string <code>rollOut</code>. • menuItem The selected menu item. Access the menu item properties as: <code>eventobj.menuItem.attributes.attribName</code> |
| rollOver | | Event | <p>Broadcast when the user moves the mouse pointer over a MenuBar item.</p> <p>The event object contains the following properties:</p> <ul style="list-style-type: none"> • target Contains a reference to the MenuBar control. • type Contains the string <code>rollOver</code>. • menuItem The selected menu item. Access the menu item properties as: <code>eventobj.menuItem.attributes.attribName</code> |

MenuItem syntax

The `<menuItem>` tag defines the items in a MenuBar control or Menu control. The following table describes the properties defined by the `<menuItem>` tag:

| Property | Type | Use | Description |
|--------------|--------|----------|--|
| enabled | String | Property | Specifies whether the menu item can be selected, <code>true</code> , or not, <code>false</code> . The default value is <code>true</code> . |
| groupName | String | Property | (Required for <code>radio</code> only) Specifies the identifier that you use to associate several radio button items in a radio group, and to expose the state of a radio group from the root menu instance. |
| icon | String | Property | Specifies the linkage identifier of an image asset. This property is not available for the <code>check</code> , <code>radio</code> , or <code>separator</code> types. |
| instanceName | String | Property | Specifies the identifier that you can use to reference the menu item instance from the root menu instance. |

| Property | Type | Use | Description |
|----------|---------|----------|--|
| label | String | Property | (Required for all except <code>separator</code>) Specifies the text displayed in the control. This property is required for all item types, except <code>separator</code> . |
| selected | Boolean | Property | Specifies the initial value of a <code>check</code> or <code>radio</code> item as <code>true</code> or <code>false</code> . The default value is <code>false</code> . For all other types, the value is <code>undefined</code> . |
| type | String | Property | Specifies the type of menu item. Values are <code>separator</code> , <code>check</code> , <code>radio</code> , or <code>normal</code> . The default value is <code>normal</code> . |

CHAPTER 4

Introducing Containers

Containers provide a hierarchical structure that lets you control the layout characteristics of container children. You can use containers to control child sizing and positioning, or to control navigation among multiple child containers.

This chapter introduces the two types of containers: layouts and navigators. This chapter contains an overview of container usage, including layout rules, and examples of how to use and configure containers.

Contents

| | |
|---|-----|
| About containers | 161 |
| Using containers | 162 |
| Controlling component sizing and positioning in a container | 170 |
| Using scroll bars | 180 |
| Creating component instances at runtime | 182 |
| Configuring containers | 185 |

About containers

A *container* defines a rectangular region of the Macromedia Flash Player drawing surface. Within a container, you define the components, both controls and containers, that you want to appear within the container. Components defined within a container are called *children* of the container.

At the root of a Macromedia Flex application is a single container, called the *Application* container, that represents the entire Flash Player drawing surface. This Application container holds all other containers, which can represent dialog boxes, panels, and forms.

A container has predefined rules to control the layout of its children, including sizing and positioning. Flex defines layout rules to simplify the design and implementation of rich Internet applications, while also providing enough flexibility to let you create a diverse set of applications.

One advantage of having predefined layout rules is that your users will soon grow accustomed to them. That is, by standardizing the rules of user interaction, your users will not have to think about how to navigate the application, but can instead concentrate on the content that the application offers.

Another advantage is that you do not have to spend time defining navigation and layout rules as part of the design process. Instead, you can concentrate on the information that you want to deliver, and the options that you want to provide for your users, and not worry about implementing all the details of user action and application response. In this way, Flex provides the structure that lets you quickly and easily develop an application with a rich set of features and interactions.

If you want a greater level of control over sizing and positioning, Flex provides the *Canvas* container. This container has no built-in layout rules, but lets you explicitly set the position and size of its children. For more information, see [“Canvas layout container” on page 206](#).

Layout containers and navigator containers

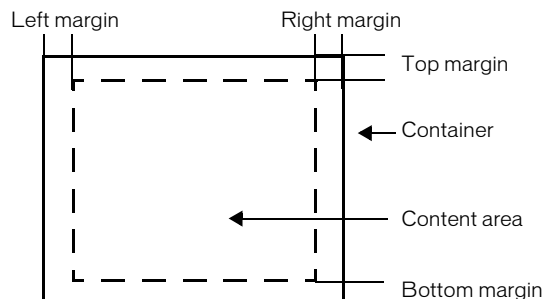
Flex defines two types of containers:

Layout containers Control the sizing and positioning of the child controls and child containers defined within them. For example, a Grid layout container sizes and positions its children in a layout similar to an HTML table. For more information, see [Chapter 6, “Using Layout Containers,” on page 205](#).

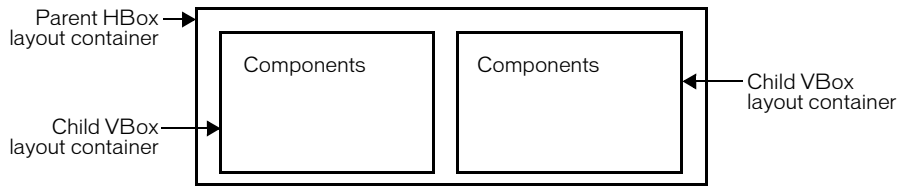
Navigator containers Control user movement, or navigation, among multiple child containers. The individual child containers, not the navigator, control the layout and positioning of their children. For example, an Accordion navigator container lets you construct a multipage form from multiple Form layout containers. For more information, see [Chapter 7, “Using Navigator Containers,” on page 247](#).

Using containers

The rectangular region of a container encloses its *content area*, the area that contains its child components. The size of the region around the content area is defined by the container margins and the width of the container border. A container has top, bottom, left, and right margins, each of which you can set to a pixel width. The border styles are none, inset (2 pixels wide), outset (2 pixels wide), and solid (1 pixel wide). The following figure shows a container and its content area, margins, and borders:



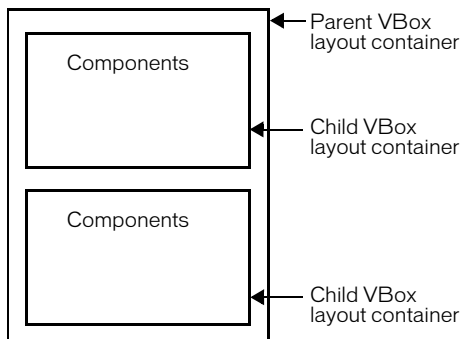
Although you can create an entire Flex application using a single container, typical applications use multiple containers. For example, the following figure shows an application that uses three layout containers:



In this example, the two VBox (vertical box) layout containers are nested within an HBox (horizontal box) layout container and are referred to as children of the HBox container.

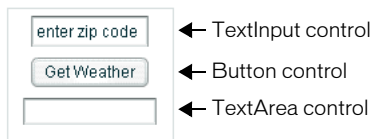
The HBox layout container arranges its children in a single horizontal row and oversees the sizing and positioning characteristics of the VBox containers. For example, you can control the distance, or gap, between children in a container using the `horizontalGap` and `verticalGap` properties.

A VBox container arranges its children in a single vertical stack, or column, and oversees the layout of its own children. The following figure shows the same example, except that the outermost container has been changed to a VBox layout container:



In this example, since the outer container is a VBox layout container, it arranges its children in a vertical column.

The primary use of a layout container is to arrange its children, where the children are either controls or other containers. The following example shows a simple VBox container that has three child components:



In this example, a user enters a ZIP code into the TextInput control, and then clicks the Button control to see the current temperature for the specified ZIP code in the TextArea control.

Flex supports form-based applications through its Form layout container. In a Form container, Flex can automatically align labels, uniformly size TextInput controls, and display input error notifications. The following figure shows an example of a Flex Form container:

Billing Information

First Name

Last Name

Address

City / State

ZIP Code

Country

Form containers can take advantage of the Flex validation mechanism to detect input errors before the user submits the form. By detecting the error, and letting the user correct it before submitting the form to a server, you eliminate unnecessary server connections. The Flex validation mechanism does not preclude you from performing additional validation on the server. For more information on Form containers, see [“Form layout container” on page 215](#).

Navigator containers, such as the TabNavigator and Accordion containers, have built-in navigation controls that let you organize information from multiple child containers in a way that makes it easy for a user to move through it. The following figure shows an example of an Accordion container:

1. Shipping Address

First Name

Last Name

Address

City

Phone

State

Zip Code

2. Billing Address

3. Credit Card Information

4. Submit Order

Accordion buttons

You use the Accordion buttons to move among the different child containers.

Accordion containers support the creation of multistep procedures. The preceding figure shows an Accordion container that defines four panels of a complex form. To complete the form, the user enters data into all four panels. Accordion containers let users enter information in the first panel, click the Accordion button to move to the second panel, and then move back to the first if they want to edit the information. For more information, see [“Accordion navigator container” on page 263](#).

Flex containers

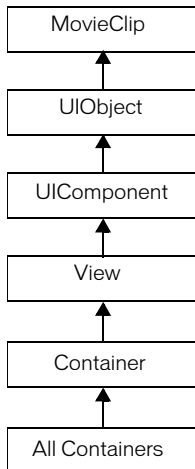
The following table describes the Flex containers:

| Container | Type | Description | For more information |
|--|-----------|--|---|
| Accordion | Navigator | Organizes information in a series of child panels, where one panel is active at any time. | “Accordion navigator container” on page 263 |
| Box (HBox and VBox) | Layout | Displays content in a uniformly spaced row or column. An HBox container horizontally aligns its children; a VBox container vertically aligns its children. | “Box layout container” on page 209 |
| Canvas | Layout | Defines a container that lets you explicitly position and size its children. | “Canvas layout container” on page 206 |
| ControlBar | Layout | Holds components shared by the other children in a Panel container. | “ControlBar layout container” on page 211 |
| DividedBox (HDividedBox and VDividedBox) | Layout | Lays out its children horizontally or vertically, much like a Box container, except that it inserts an adjustable divider between each child. | “DividedBox layout container” on page 212 |
| Form | Layout | Arranges its children in a standard form format. | “Form layout container” on page 215 |
| Grid | Layout | Arranges children as rows and columns of cells, much like an HTML table. | “Grid layout container” on page 229 |
| LinkBar | Navigator | Defines a row of Link controls designating a series of link destinations, often used with a ViewStack container. | “LinkBar navigator container” on page 253 |
| Panel | Layout | Displays a title bar, caption, border, and its children. | “Panel layout container” on page 234 |
| TabBar | Navigator | Defines a horizontal row of tabs, often used with a ViewStack container. | “Accordion navigator container” on page 263 |
| TabNavigator | Navigator | Displays a container with tabs to let users switch between different content areas. | “TabNavigator container” on page 259 |
| Tile | Layout | Defines a layout that arranges its children in multiple rows or columns. | “Tile layout container” on page 237 |

| Container | Type | Description | For more information |
|-------------|-----------|--|---|
| TitleWindow | Layout | Displays a modal window that contains a title bar, caption, border, close button, and its children. The user can move and resize it. | “TitleWindow layout container” on page 239 |
| ViewStack | Navigator | Defines a stack of panels that displays a single panel at a time. | “ViewStack navigator container” on page 248 |

Class hierarchy for containers

Flex containers are implemented as a hierarchy in an ActionScript class library, as the following figure shows:



All containers are derived from the ActionScript classes MovieClip, UIObject, UIComponent, View, and Container, and therefore inherit the properties of their parent classes. For a complete reference, see *Flex ActionScript and MXML API Reference*.

Container example

The following figure shows an example Flex application that uses a single VBox container with three controls:



The following MXML code creates this example:

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

  <mx:Panel title="My Application" >

```

```

<mx:VBox id="myVBox" borderStyle="solid" horizontalAlign="center"
  marginLeft="10" marginRight="10" marginTop="5" marginBottom="5" >

  <!-- TextInput control for ZIP code. -->
  <mx:TextInput id="myinput" text="enter zip code" />

  <!-- Button control to get weather and update the TextArea control. -->
  <mx:Button id="mybutton" label="GetWeather" click="getTemp()" />

  <!-- TextArea field for results of the web service call. -->
  <mx:TextArea id="mytext" height="20" />
</mx:VBox>
</mx:Panel>
</mx:Application>

```

The code for the `getTemp()` function is not shown in this example.

The following figure shows the same example, this time implemented using an `HBox` container:



The only difference in these two examples is the container type and the increased width of the `Application` container because of its horizontal layout, as the following code shows:

```

<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

  <mx:HBox id="myHBox" borderStyle="solid" marginLeft="10" marginRight="10"
    marginTop="5" marginBottom="5" verticalAlign="middle" >

    ...

  </mx:HBox>

</mx:Application>

```

Handling events when creating components

All containers support a `childrenCreated` event, which Flex triggers after Flex creates the child controls inside a container. You can use this event to perform any final configuration of a container's children.

This event is useful with a container that is an immediate child of a navigator container, such as the `ViewStack`, `TabNavigator`, and `Accordion` navigator containers. For example, a `ViewStack` container creates all its immediate child containers, and all the children of its first visible child container. For the first visible child container, `childrenCreated` gets broadcast. Then, as the user moves to each additional child of the `ViewStack`, the event gets dispatched for that container.

The following example defines an event handler for the `childrenCreated` event, which is broadcast when the user first navigates to `pane2` of the `ViewStack` navigator container:

```

<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:Script>
    <![CDATA[

```

```

        function pane2_childrenCreated(event):Void {
            ...
        }
    ]]>
</mx:Script>

<mx:ViewStack>
    <mx:HBox id="panel">
        ...
    </mx:HBox>
    <mx:HBox id="pane2" childrenCreated="pane2_childrenCreated(event)">
        ...
    </mx:HBox>
</mx:ViewStack>
</mx:Application>

```

For more information on the ViewStack navigator container, see [“ViewStack navigator container” on page 248](#).

Disabling containers

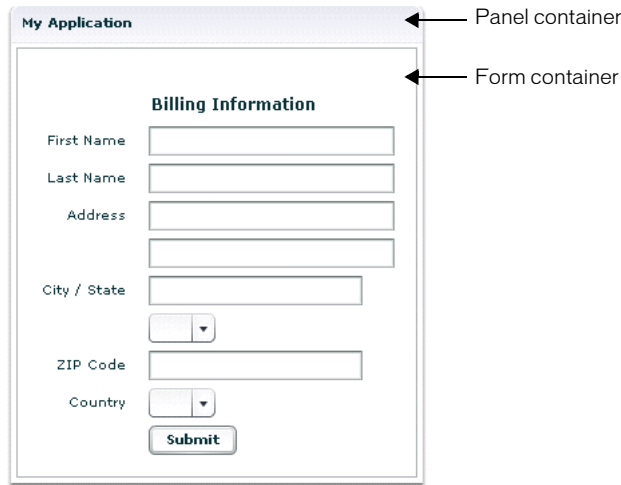
All containers support the `enabled` property. By default, this property is set to `true` to enable user interaction with the container and with the container’s children. If you set `enabled` to `false`, Flex dims the color of the container and of all of its children, and blocks user input to the container and to all of its children.

Using the Panel container

One container that you often use in a Flex application is the Panel container. The Panel container consists of a title bar, a caption, a status message, a border, and a content area for its children. Typically, you use a Panel container to wrap self-contained application modules. For example, you can define several Panel containers in your application where one Panel container holds a form, a second holds a shopping cart, and a third holds a shopping catalog.

To see an example application using multiple Panel containers, run the FlexStore sample application. The FlexStore application included in the `samples.war` file uses several different Panel containers to define its layout.

The following figure shows a Panel container with a Form container as its child:



For more information on the Panel container, see [“Panel layout container” on page 234](#).

You can also define a ControlBar control as part of a Panel container. A ControlBar control defines an area at the bottom of the Panel container, below any children in the Panel container.

You can use the ControlBar container to hold components that might be shared by the other children in the Panel container. For example, you can use the ControlBar container to display the subtotal of a shopping cart, where the shopping cart is defined in the Panel container. For a product catalog, the ControlBar container can hold the Flex controls to specify quantity and to add an item to a shopping cart. For more information on the ControlBar container, see [“ControlBar layout container” on page 211](#).

Defining a default button

You use the `defaultButton` property of a container to define a default Button control within a container. Pressing the Enter key while focus is on any control activates the Button control just as if it was explicitly selected.

For example, a login form displays TextInput controls for a user name and password and a submit Button control. Typically, the user types a user name, tabs to the password field, types the password, and presses the Enter key to submit the login information without explicitly selecting the Button control. To define this type of interaction, set the `defaultButton` to the `id` of the submit Button control, as the following example shows:

```
<mx:Form defaultButton="mySubmitButton" >
  <mx:FormItem label="Username">
    <mx:TextInput id="username" width="100"/>
  </mx:FormItem>
  <mx:FormItem label="Password">
    <mx:TextInput id="password" width="100" password="true"/>
  </mx:FormItem>
  <mx:FormButton id="mySubmitButton" label="Submit" />
</mx:Form>
```

```
<mx:Button id="mySubmitButton" label="Login" click="submitLogin(event)"/>
</mx:FormItem>
</mx:Form>
```

Note: The ComboBox control has a special meaning for the Enter key. When the drop-down list of a ComboBox control is open, pressing Enter selects the currently highlighted item in the ComboBox control; it does not activate the default button.

Controlling component sizing and positioning in a container

A container controls the layout of its children using a well-defined set of rules. The layout rules are a combination of the rules for the children and the rules of the parent container. Layout rules mean that you do not have to worry about positioning container children. Instead, you concentrate on building the logic of your application and let Flex control the layout.

Each container has its own rules for controlling layout. For example, the VBox container lays out its children in a single column. A Grid container lays out its children in rows and columns of cells.

Although containers have built-in layout rules, each container also has properties that you use to configure aspects of layout. For example, you can use the `verticalGap` and `horizontalGap` properties of a Tile container to set the spacing between children, and the `direction` property to specify either a row or column layout.

If you require total control over the sizing and positioning of container children, use the Canvas container. The Canvas container does not have any predefined layout rules. Instead, you explicitly set the position of its children using the `x` and `y` properties of each child, and set the size using the `height` and `width` properties of each child.

The Flex two-pass layout algorithm

Flex uses a two-pass algorithm to determine the size and position of each component in an application:

Measurement pass Calculates the preferred size of every component in the application, starting from the innermost component and working out toward the outermost container.

Every component has a preferred height and width. For some components, Flex calculates that value based on properties of the component. For example, the preferred size of a Button control is large enough to hold the Button label. For a VBox container, the preferred size is large enough to hold all of its children based on the preferred sizes of the children.

Layout pass Lays out your application, including moving and resizing any components, starting from the outermost container and working in toward the innermost component.

By default, components cannot be resized from their preferred size. During the layout pass, Flex determines whether any components are resizable. Flex can enlarge or shrink resizable components during the layout pass.

General positioning and sizing rules for all containers

Each container uses different rules for sizing itself and its children; however, the following rules are true for all containers:

1. During the measurement pass, Flex sizes children to their preferred size, the size used for the objects if no explicit sizes are specified. If the `width` or `height` property has an explicit pixel value, that value is used instead of the preferred width or preferred height.
Note: When sizing and positioning components, Flex does not distinguish between visible and invisible components. That means an invisible component is sized and positioned as if it were visible. However, setting a component's `height` and `width` properties to 0 makes it take up no space. You can then set its `height` and `width` properties to `undefined` at runtime, which forces Flex to recalculate its size and perform another layout pass. Or, you can set the component to an explicit size.
2. To allow Flex to resize a child during the layout pass, specify a percentage value for its `height` and/or `width` properties. A container can have fixed-size children, flexible children, or a mix of the two.
3. You can include values for `minWidth`, `maxWidth`, `minHeight`, and `maxHeight` to set sizing limits. By default, Flex sets most components' minimum height and width to 0. Flex does not define a default maximum size for most components, so they are effectively unbounded.
4. Setting the `height` or `width` properties to a specific integer value for a child means that you are setting an explicit size for the child, and that Flex cannot resize it in the corresponding direction.
Note: If you read the `width` and `height` properties of a component at runtime using ActionScript, you get the values of the component's current size in pixels, regardless of whether you set them to explicit values, set them to percentage values, or don't set them at all. If you set percentage values, read `percentWidth` and `percentHeight` to retrieve the percentages.
5. If a child is larger than its parent container, the parent clips the child at the parent's boundaries, meaning that the parent does not draw a child outside of itself, and displays scroll bars. Set the `clipContent` property to `false` to configure a parent to let the child extend past its boundaries.

Setting the Application container size

The first aspect of sizing in a Flex application is setting the size of the Application container. The Application container determines the boundaries of your application in Flash Player. By default, the Application container is sized to 100%, meaning it takes up the entire browser window.

You set the Application container size using the `<mx:Application>` tag, as the following example shows:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    height="100" width="150">
    ...
</mx:Application>
```

In this example, you set the Application container size to 100 x 150 pixels. Anything in the application larger than this window is clipped at the window boundaries. Therefore, if you define a 200 pixel x 200 pixel DataGrid control, it is clipped, and the Application container displays scroll bars.

For more information on sizing the Application container, see [Chapter 5, “Using the Application Container,” on page 191](#).

Calculating the preferred size of a component

In the measurement pass, Flex calculates the preferred size of every component in your application, starting from the innermost component and working out toward the Application container. Flex starts at the innermost component because the preferred size of many containers is based on the children that are defined within them.

Note: During the measurement pass, Flex sets the default minimum size of most components to 0. You can use the `minWidth` and `minHeight` properties to set your own values for the minimum size. The default maximum size is typically undefined, which means that there is no limit to a component's maximum size. You can use the `maxWidth` and `maxHeight` properties to set your own values for the maximum size.

The following example defines a `TextInput` control within an `HBox` container:

```
<mx:HBox id="myHBox">
  <mx:TextInput id="myInput" text="Enter ZIP code" />
</mx:HBox>
```

In this example, Flex first calculates the preferred size of the `TextInput` control, then of the parent `HBox` container. The preferred size of the `TextInput` control is based on the pixel length of the string value of its `text` property. The preferred size of an `HBox` container is a size large enough to hold all of its children sized to their preferred size, plus any margins. Therefore, Flex must already have calculated the preferred size of a container's children before it can calculate the container's preferred size.

Each component has rules for determining its preferred size. For more information, see the description of each component.

Specifying an explicit size

You use the `width` and `height` properties of a component to explicitly set its size, as follows:

```
<mx:HBox id="myHBox" >
  <mx:TextInput id="myInput" label="Enter the zip code"
    width="200" height="40" />
</mx:HBox>
```

During the measurement pass, Flex sets the preferred size of the component to the values of the `width` and `height` properties. However, Flex cannot resize an explicitly sized component during the layout pass.

You can also use the `width` and `height` properties of the parent `HBox` container to explicitly set its size, as follows:

```
<mx:HBox id="myHBox" width="150" height="150" >
  <mx:TextInput id="myInput" label="Enter the zip code"
    width="200" height="40" />
</mx:HBox>
```

In this example, because you set the `TextInput` control to a size that is larger than its parent `HBox` container, Flex clips the `TextInput` control at the container boundaries and displays scroll bars so that you can scroll the container to see its children.

If you set the `clipContent` property for the parent `HBox` container to `false`, the container lets the `TextInput` control extend beyond its boundaries and no scroll bars appear, as the following example shows:

```
<mx:HBox id="myHBox" width="150" height="150" clipContent="false" >
  <mx:TextInput id="myInput" label="Enter the zip code"
    width="200" height="40" />
</mx:HBox>
```

Making a component resizable

By default, a component is not resizable. However, every Flex component supports flexible sizing through percentage `width` and `height` values.

Flex attempts to size each resizable component to the percentage of its parent container that is specified in the `width` and `height` properties. If the parent container also has fixed-size children and there is not enough space left to satisfy the flexible component percentage requests, the available space is divided between the flexible components in proportion to the requested percentages.

For example, suppose that 50% of a parent is empty after placing the fixed-size components, one flexible component requests 20% of the parent, and another flexible component requests 60%; the first flexible component is sized to 12.5% of the parent container and the second to 37.5% of the parent container, provided there are no gaps between components.

Flex respects the defined margins and gaps between components when calculating the size of flexible components, so a component set to 100% may not entirely fill its parent container. For instance, a `VBox` container set to consume 100% of its parent `Application` container's space will leave 24-pixel-wide top, bottom, left, and right margins, unless you explicitly change the margin settings of the `Application` container.

Resizing children

To resize children, Flex does the following:

1. Determines the size of the parent container if it is not explicitly specified. For more information on calculating the size of containers, see [“Determining the size of containers” on page 178](#).
2. Calculates the preferred size of all children. Assigns a preferred size of 0 to flexible children.
3. Reserves the required number of pixels for the fixed size components.
4. Determines the requested size of each flexible component based on the size of the parent container. Flex rounds noninteger calculated values down to the nearest integer. You can include values for `minWidth`, `maxWidth`, `minHeight`, and `maxHeight` to set sizing limits. If the required number of pixels are available, they are allotted to the flexible children and the process ends.
5. If the parent container does not have enough empty space to accommodate the percentage requests, Flex divides the available space in proportion to the specified percentages. Flex rounds noninteger calculated values down to the nearest integer.

Example with no widths specified

Consider the following example, in which none of the children of the HBox container specify a width value:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:HBox width="400" borderStyle="solid" >
    <mx:Button label="Label 1" />
    <mx:Button label="Label 2" />
    <mx:Button label="Label 3" />
  </mx:HBox>
</mx:Application>
```

In this example, none of the children are resizable, and Flex draws this application as the following figure shows:



Notice the empty space to the right of the third button.

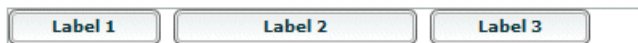
Example with two resizable children

The following example makes the first two buttons resizable:

```
<mx:HBox width="400">
  <mx:Button label="Label 1" width="25%" />
  <mx:Button label="Label 2" width="40%" />
  <mx:Button label="Label 3" />
</mx:HBox>
```

In this example, assume that the preferred width of the third button is 100 pixels, so the HBox container has 300 pixels of extra space remaining. The HBox container has no margins by default, but it does require an 8-pixel horizontal gap between each component. Because this application has three components, 16 pixels are used for these gaps. The first button requests 25% of the HBox, or 100 pixels. The second button requests 40% of 400 pixels, or 160 pixels. There is enough free space to accommodate this, so Flex sizes the buttons to 100 and 160 pixels, respectively, and the children are laid out with these final width values. There is still unused space to the right of the third button because the flexible controls did not encompass the entire unclaimed 284 pixels.

Flex draws the application as follows:



Now change the percentage values requested to 50% and 40%, respectively:

```
<mx:HBox width="400">
  <mx:Button label="Label 1" width="50%" />
  <mx:Button label="Label 2" width="40%" />
  <mx:Button label="Label 3" />
</mx:HBox>
```

In this example, the first button requests 50% of the HBox, or 200 pixels. The second button still requests 40%, or 160 pixels. However, the HBox only has 284 pixels free after reserving 100 pixels for the fixed-width button and 16 pixels for the gap between components. Flex divides that space proportionally between the two buttons, giving 5/9 of the space, or 157.78 (rounded down to 157) pixels, to the first button and 4/9 of the space, or 126.22 (rounded down to 126) pixels, to the second button.

Flex draws the application as follows:



Example with minimum width specified

You can also include the `minWidth`, `minHeight`, `maxWidth`, and `maxHeight` properties with a resizable component to constrain its size. Consider the following example:

```
<mx:HBox width="400">
  <mx:Button label="Label 1" width="50%" />
  <mx:Button label="Label 2" width="40%" minWidth="150"/>
  <mx:Button label="Label 3" />
</mx:HBox>
```

Assume again that the preferred width of the fixed-size button is 100 pixels, so there are 284 pixels of space available for the flexible buttons after accounting for the gap between components. Flex repeats the calculations described in the previous example, determining that the first button should be 157 pixels wide and the second 122 pixels. However, the second button must be at least 150 pixels wide. Flex assigns that button the set width of 150 pixels and recalculates the size of the other flexible button. The button requests 200 pixels for its width, but there are only 134 pixels available in the container. Because it is the final remaining unsized component, the remainder of the space is given to the first button, now sized to 134 pixels.

Flex draws the application as follows:



Example with scroll bars

If Flex cannot fit all of the components into the container, it will use scroll bars. This can happen even with flexible components. Consider the following example:

```
<mx:HBox width="400">
  <mx:Button label="Label 1" width="50%" minWidth="150"/>
  <mx:Button label="Label 2" width="40%" minWidth="150"/>
  <mx:Button label="Label 3" />
</mx:HBox>
```

Assume again that the preferred width of the fixed-size button is 100 pixels, so there are 284 pixels of space available for the flexible buttons after accounting for the gap between components. Flex repeats the calculations described in the previous example, and determines that the first button should be 157 pixels wide and the second 122 pixels. However, both buttons must be at least 150 pixels wide. Flex assigns those buttons the set width of 150 pixels, even though the HBox container only had 284 pixels free. The HBox container uses scroll bars to provide access to its contents because they now consume more space than the container itself.



Notice that the addition of the scroll bar doesn't increase the height of the container. So, unless you explicitly set the height property for the HBox container or allow the HBox container to resize by setting a percentage width, the scroll bar will overlap the buttons. Remember that changing the height of this container will cause other containers in your application to move and resize according to their own sizing rules. The following example adds an explicit height and permits you to see the buttons and the scroll bar simultaneously:

```
<mx:HBox width="400" height="42">
  <mx:Button label="Label 1" width="50%" minWidth="150"/>
  <mx:Button label="Label 2" width="40%" minWidth="150"/>
  <mx:Button label="Label 3" />
</mx:HBox>
```



Alternately, you can set the `hScrollPolicy` to `on`. This reserves space for the scroll bar during the initial layout pass so it fits without overlapping the buttons or setting an explicit height. This also correctly handles the situation where the scroll bars change their size when you change skinning or styles.

Example with margins

There may be situations where you want your containers to have margins. Some containers, like the Application container, have margins by default. If your application has margins, the necessary pixels will be reserved before any flexible components are sized. Consider the following example:

```
<mx:HBox width="400" marginLeft="5" marginRight="5" horizontalGap="5" >
  <mx:Button label="Label 1" width="50%" />
  <mx:Button label="Label 2" width="40%" minWidth="150"/>
  <mx:Button label="Label 3" />
</mx:HBox>
```


Assume again that the preferred width of the fixed-size button is 100 pixels. You have explicitly set all horizontal margins and gaps to 5 pixels wide, so your application will reserve 5 pixels for the left margin, 5 pixels for the right margin, and 10 pixels total for the two gaps between components, which leaves 280 pixels free for the two flexible components. Flex repeats the calculations described in the previous example, determining that the first button should be 157 pixels wide and the second 122 pixels. However, the second button must be at least 150 pixels wide. Flex assigns that button the set width of 150 pixels, and recalculates the size of the other flexible button. The button requests 200 pixels for its width, but there are only 130 pixels available in the container. Because it is the final remaining unsized component, the remainder of the space is given to the first button, now sized to 130 pixels.

Flex draws the application as follows:



Dealing with components that don't fit in their container

If you change the width of the HBox to 250 pixels wide, assuming the preferred size of each button is 100 pixels, the buttons do not fit within the HBox container as fixed-width components. By default, the buttons are clipped at the container boundaries, and the container displays scroll bars. If you set `clipContent` to `false`, the buttons extend beyond the boundaries of the container.

Alternately, Flex shrinks flexible children to their minimum sizes. By default, Flex sets the minimum height and width of most components to 0. You should set these properties to nonzero values to ensure that they remain readable.

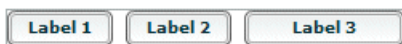
Example with components that shrink

The following example shows that the flexible buttons are smaller than they would be if no width were specified:

```
<mx:HBox width="250">
  <mx:Button label="Label 1" width="40%" minWidth="50" />
  <mx:Button label="Label 2" width="40%" minWidth="50" />
  <mx:Button label="Label 3" />
</mx:HBox>
```

In this example, 100 of the 250 pixels are reserved for the fixed-width button, which leaves 134 pixels for the flexible buttons after accounting for the gaps between components. Each flexible button desires 40% of the parent's space, or 100 pixels, but there isn't enough space. The available 134 pixels are split evenly between the two components because they were set to the same percentage value, yielding widths of 67 pixels each.

Flex draws the application as follows:



Determining the size of containers

By default, the size of a container is calculated by determining the preferred size of its children. However, flexible children have no independent preferred size; their size is specified as a set portion of the parent container. On the surface, this seems like circular logic with two distinct values, each calculated from the value of the other. However, this contradiction does not really exist because Flex does not evaluate the size dependence of children until after their parent's size is set. Thus, at the time the container is sized, each of its child components has a set, nonzero preferred size that is used in the container-sizing operation (step 1 of the sizing operation described in [“Making a component resizable” on page 173](#)). Only after these calculations are complete are the preferred sizes of all flexible children set to 0 (in step 2 of the sizing operation described in [“Making a component resizable” on page 173](#)).

If a container has flexible children, Flex calculates its preferred size using the following rules:

1. If the container declares an explicit size value, that size is used as the preferred size.
2. If the container declares a percentage size value, its preferred width is calculated according to the normal flexible sizing rules using the size declarations of its parent and siblings. If the container's parent declares a percentage size value, Flex examines the next container outward. This chain is followed outward until Flex encounters a container with a fixed size. If necessary, this chain propagates out to the Application container, which is sized to 100% of the browser size by default. When Flex encounters explicit size values in a parent container, the chain is followed inward, calculating explicit sizes for each set of containers using the preceding rules until the original parent container has an explicit size.
3. If the container has no size values specified, the preferred size of each of the children is used to calculate the preferred size of the parent container just as if its children were all fixed-width components. Flex does not set the preferred size of the flexible children to 0 until after it determines the parent size.

Using the Spacer control to control layout

Flex includes a Spacer control that helps you lay out children within a parent container. The Spacer control is invisible but it does allocate space within its parent.

In the following example, you use a flexible Spacer control to push the Button control to the right so that it is aligned with the right edge of the HBox container:

```
<mx:HBox width="400">
  <mx:Image source="Logo.jpg" />
  <mx:Label text="Company XYZ" />
  <mx:Spacer width="100%" />
  <mx:Button label="Close" />
</mx:HBox>
```

In this example, the Spacer control is the only resizable component in the HBox container. During the measurement pass, Flex sets all components to their preferred size; during the layout pass, Flex sizes the Spacer control to occupy all additional space in the HBox container. By expanding the Spacer control, Flex pushes the Button control to the right edge of the container.

You can use all sizing and positioning properties with the Spacer control, such as `width`, `height`, `maxWidth`, `maxHeight`, `minWidth`, `minHeight`, `preferredWidth`, and `preferredHeight`.

Triggering a layout at runtime

Flex performs a layout pass on your application when your application initializes. You can also cause Flex to perform a layout pass during application execution in the following situations:

- The application changes any of the following properties in `ActionScript`: `x`, `y`, `width`, `height`, `minWidth`, `minHeight`, `preferredWidth`, `preferredHeight`, `maxWidth`, `maxHeight`, `scaleX`, and `scaleY`. Calling the `setSize()` method will also trigger a layout pass.
- A change affects the calculated width or height of a component, such as the label text for a `Button` control that is being modified.

Disabling container layout

By default, Flex updates the size of a container and the layout of the container's children when a new child is added or removed from the container, when a child is resized, and when a child is moved. For example, if your application contains functionality to change the size of a component, Flex updates the layout of the container to reposition its children, based on the new size of the child.

You can also use effects, such as the `Move` and `Zoom` effects, to modify the size or position of a child in response to a user action. For example, you might define a child so that when the user selects it, the child moves to the top of the container and doubles in size. These effects modify the `x` and `y` properties of the child as part of the effect.

However, the container controls the layout of its children, so it ignores the values of the `x` and `y` properties of its children during a layout update. Therefore, the layout update cancels any modifications to the `x` and `y` properties performed by the effect, and the child does not remain in its new location.

To disable Flex from performing layout updates that conflict with the requested action of your application, you can set the `autoLayout` property of a container to `false`. Flex defines the `autoLayout` property in the `Container` class, and all containers inherit it. Its default value is `true`, which enables Flex to update layouts.

Even when you set the `autoLayout` property of a container to `false`, Flex updates the layout of a container when you add a child to it or remove a child from it. Application initialization, deferred instantiation, and the `<mx:Repeater>` tag add or remove children, so layout updates always occur during these processes, regardless of the value of the `autoLayout` property. Therefore, during container initialization, Flex defines the initial layout of the container children regardless of the value of the `autoLayout` property.

For example, the following code disables layout updates for a `VBox` container, but still uses the rules of the `VBox` container to determine the initial position of its children:

```
<mx:VBox autoLayout="false" width="200" height="200">
  <mx:Button/ >
  <mx:Button id="btn" click="btn.x += 10"/>
  <mx:Button id="btn2" creationComplete="btn2.x = 100; btn2.y = 75"/>
</mx:VBox>
```

In this example, Flex positions the first Button control according to the rules of the VBox container. Flex positions the second Button control according to the rules of the VBox container, and shifts it 10 pixels to the right when a user selects it. The third Button control uses an event to position it relative to the location that the VBox container initially determines for it.

Since setting the `autoLayout` property of a container to `false` prohibits Flex from updating a container's layout after a child moves or resizes, you should only set it to `false` when absolutely necessary. You should always test your application with the `autoLayout` property set to the default value of `true`, and set it to `false` only as necessary for the specific container and specific actions of the children in that container.

For more information on effects, see [Chapter 18, “Using Behaviors,”](#) on page 453.

Using scroll bars

Flex containers support scroll bars, which let you display an object that is larger than the available screen space, as the following figure shows:



In this figure, you use an HBox container to let users scroll an image, rather than rendering it at its full size. The following code defines the HBox container shown in this figure:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    borderStyle="solid" height="600" width="600">

    <mx:HBox width="75" height="75" >
        <mx:Image source="mm.jpg" />
    </mx:HBox>
</mx:Application>
```

In this example, you explicitly size the HBox container to 75 x 75 pixels, a size much smaller than the imported image. If you omit the sizing restrictions on the HBox container, it will attempt to use its preferred size, which is a size large enough to hold the image.

By default, Flex draws scroll bars only when the contents of a container are larger than that container. To force the container to draw scroll bars, you can set the `hScrollPolicy` and `vScrollPolicy` properties to `on`.

Working with scroll bars

The setting of the `hScrollPolicy` and `vScrollPolicy` properties of a container controls the display of scroll bars. By default, both properties are set to `auto` to configure Flex to include scroll bars only when necessary. You can set these properties to `on` to configure Flex to always include them, or set them to `off` to configure Flex to never include them.

Note: If `clipContent` is `false`, a container allows its child to extend past its boundaries. Therefore, no scroll bars are necessary, and Flex will never display them, even if you set `hScrollPolicy` and `vScrollPolicy` to `on`.

The properties `hLineScrollSize` and `vLineScrollSize` determine how many pixels to scroll when the user selects the scroll bar arrows. The default is 5 pixels. The properties `hPageScrollSize` and `vPageScrollSize` determine how many pixels to scroll when the user selects the scroll bar track. The default value is 20 pixels.

Your configuration of scroll bars can affect the layout of your application. For example, if you set the `hScrollPolicy` and `vScrollPolicy` properties to `on`, the container always includes scroll bars, even if they are not necessary. Since each scroll bar is 16 pixels wide, turning them on is the same as increasing the size of the right and bottom margins of the container by 16 pixels.

If you keep the default `hScrollPolicy` and `vScrollPolicy` property values of `auto`, then Flex performs layout just as if the properties are set to `off`. That is, the scroll bars are not counted as part of the layout.

However, suppose you have an `HBox` container that is 30 pixels high and 100 pixels wide. Initially, it contains two buttons that are each 22 pixels high and 40 pixels wide. The children are fully contained inside the `HBox`, and no scroll bars are necessary.

You then add a third button, which is the same size as the other buttons. Now the width of the children exceeds the width of the `HBox`, so Flex adds a horizontal scroll bar at the bottom of the container.

The horizontal scroll bar is 16 pixels high, which reduces the height of the content area of the container from 30 pixels to 14 pixels. That means your `Button` controls, which are 22 pixels high, are now too tall for the `HBox`, and Flex clips the `Button` controls.

You can use the `repeatDelay` and `repeatInterval` styles to control scroll bars. The `repeatDelay` style specifies the number of milliseconds to wait after the user selects a scroll button before scrolling. The `repeatInterval` style specifies the number of milliseconds to wait between each scroll.

Note: Even though these styles are defined in the `SimpleButton` class, they apply to scroll bars.

The following example uses these styles with an `HBox` container:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    borderStyle="solid" height="600" width="600">

    <mx:Style>
        global{
            repeatInterval: 1000;
            repeatDelay: 2000;
```

```

    }
</mx:Style>

    <mx:HBox width="75" height="75" >
        <mx:Image source="mm.jpg" />
    </mx:HBox>
</mx:Application>

```

Creating component instances at runtime

You typically use MXML to lay out the user interface of your application, and use ActionScript for event handling and runtime control of the application. You can also use ActionScript to create component instances at runtime. For example, you can use MXML to define an empty Accordion container. Then, in ActionScript, add additional panels to the Accordion container in response to user actions.

To create a component instance at runtime, you create it as a child of a parent container by calling the `View.createChild()` method on the parent container. This method has the following signature:

```
createChild(classOrSymbol:var, undefined, initProps:Object) : MovieClip;
```

where:

- *classOrSymbol* Specifies a class name. This argument is required.

You must specify the fully qualified class name of the component that you want to create. For example, if your application has an HBox container named `myHB`, the following code creates a Button control within that container:

```
myHB.createChild(mx.controls.Button);
```

If you first import the class that you want to create, you only need to specify the class name, as the following example shows:

```
import mx.controls.Button;
myHB.createChild(Button);
```

- *undefined* This argument is not used by Flex so you set it to `undefined`. This argument is optional, and is necessary only as a placeholder if you want to specify the *initProps* argument.
- *initProps* Specifies an optional object containing initialization properties. This argument is optional.

The *initProps* argument lets you pass initialization properties to the component you are creating. You can pass to it the same properties that you can set in MXML for the component. For example, to create a Button control with the label `New Button` and a height and width of 100 pixels, you pass an *initProps* argument as the following example shows:

```
import mx.controls.Button;
myHB.createChild(Button, undefined,
    {label:'New Button', width:100, height:100});
```

The `addChild()` method returns a `MovieClip` object representing the new component. You often assign the return value of the `addChild()` method to a variable so that you can manipulate the component programmatically. For example, the following code creates a `Button` control:

```
import mx.controls.Button;

var myNewButton:Button;
myNewButton = Button(myHB.addChild(Button));
```

It is a good practice to cast the return value of the `addChild()` method to the data type of the new component so that the Flex compiler can perform type-checking on the variable if you use it to manipulate the component. For example, you can use the following code to increase the height and width of the `Button` control by 100 pixels:

```
myNewButton.height+=100;
myNewButton.width+=100;
```

You can also use the `mx.core.View` class's `destroyChild()` method to delete a control, as the following example shows:

```
myHB.destroyChild(myNewButton);
```

For additional methods that you can use with container children, see the `View` class in *Flex ActionScript and MXML API Reference*.

Creating a component as a child of an `HBox` container

The following example defines an empty `HBox` container and two `Button` controls in MXML. You use one `Button` control to add a `CheckBox` control to the `HBox` container, and one `Button` control to delete it.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

  <mx:Script>
    <![CDATA[

      // Import the Checkbox class.
      import mx.controls.CheckBox;

      // Define a variable to hold the new CheckBox control.
      var addedCheckBox:CheckBox;
      // Define a variable to track if you created a CheckBox control.
      var checkBoxCreated:Boolean = false;

      function addCB()
      {
        // Test to see if you have already created the CheckBox control.
        if(checkBoxCreated==false){
          addedCheckBox = CheckBox(myHB.addChild(CheckBox, undefined,
            {label:"New CheckBox"}));
          checkBoxCreated=true;
        }
      }

    ]]>
  </mx:Script>

  <mx:HBox>
    <mx:Button label="Add CheckBox" click="addCB()" />
    <mx:Button label="Delete CheckBox" click="deleteCB()" />
  </mx:HBox>
</mx:Application>
```

```

function delCB()
{
    // Make sure a CheckBox control exists.
    if(checkBoxCreated==true){
        myHB.destroyChild(addedCheckBox);
        checkBoxCreated=false;
    }
}
]]>
</mx:Script>

<mx:VBox >

    <mx:HBox id="myHB" borderStyle="solid" />

    <mx:Button label="add CheckBox" click="addCB()" />
    <mx:Button label="remove CheckBox" click="delCB()" />

</mx:VBox>
</mx:Application>

```

Creating a child of an Accordion container

The example in this section adds panels to an Accordion container. The Accordion container initially contains no panels. Each time you select the Add HBox button, it adds a new HBox container to the Accordion container.

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

    <mx:Script>
        <![CDATA[

            // Import HBox class.
            import mx.containers.HBox;

            //Array of created containers
            var hBoxes:Array = [];

            function addHB()
            {
                hBoxes[hBoxes.length] = HBox(myAcc.createChild(HBox, undefined,
                    {label:'my label: ' + hBoxes.length } ));
            }

            function delHB()
            {
                myAcc.destroyChild(hBoxes[(hBoxes.length-1)]);
                // Delete last Array item.
                hBoxes.pop();
            }

        ]]>
    </mx:Script>

```



```

<mx:VBox>

    <mx:Accordion id="myAcc" />

    <mx:Button label="Add HBox" click="addHB()" />
    <mx:Button label="Remove HBox" click="delHB()" />

</mx:VBox>
</mx:Application>

```

Indexing dynamically created components

The new child created by `addChild()` becomes the last child in the container. If you want to insert the new child between two existing children, use the View method `setChildIndex()` to dynamically re-index the container.

The following example makes `addedCheckBox` the second child in the container (child indexes are zero-based):

```
myHB.setChildIndex(addedCheckBox, 1);
```

To get the number of children in a given container, get the value of the container's `numChildren` property, as the following example shows:

```
var myHBChildren:Number = myHB.numChildren;
```

To get the number of children in the entire application, use the application object's `numChildren` property, as the following example shows:

```
var myAppChildren:Application = mx.core.Application.application.numChildren;
```

For more information on accessing the root application, see [“About scope” on page 325](#).

Configuring containers

Like any Flex component, you configure containers using properties, methods, styles, behaviors, events, and skins. This section contains an overview of container configuration.

Using properties and methods to configure a container

Containers inherit the properties and methods of the `MovieClip`, `UIObject`, and `UIComponent` classes. Containers also inherit the properties and methods of the `View` and `Container` classes. For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*. The following table describes the optional inherited properties:

| Property | Type | Description |
|------------------------------|---------|---|
| <code>defaultButton</code> | String | Specifies the <code>id</code> of the <code>Button</code> control to assign as the default button for the container. Pressing the Enter key while focus is on any form control activates the <code>Button</code> control just as if it was explicitly selected. |
| <code>clipContent</code> | Boolean | Specifies that the container clips its children at the container boundaries, which means that the container will not draw a child outside of itself. The default value is <code>true</code> so that the container clips its children. Set it to <code>false</code> to disable clipping. |
| <code>hLineScrollSize</code> | Number | Specifies the number of pixels to move when the user selects the left or right arrow in a horizontal scroll bar, or uses the Left or Right Arrow key. The default value is 5. |
| <code>hPageScrollSize</code> | Number | Specifies the number of pixels to move when the user clicks the scroll bar track of a horizontal scroll bar, or uses the Page Up or Page Down key. The default value is 20. |
| <code>hPosition</code> | Number | Specifies the pixel position of the slider in the horizontal scroll bar, where a value of 0 corresponds to the left end of the scroll bar. The default value is 0. |
| <code>hScrollPolicy</code> | String | Specifies when to include a horizontal scroll bar: <ul style="list-style-type: none">• <code>on</code> Causes the container to always include a horizontal scroll bar• <code>off</code> Always excludes the scroll bar• <code>auto</code> (default) Allows the container to include it only when the width of its contents is larger than the width of the container. |
| <code>icon</code> | File | Specifies the URL to an image file for an icon. Image types include JPEG, GIF, PNG, SVG image, and SWF file. You use the following format with this property: <code>icon="@Embed('relativeURL')"</code> The referenced image is packaged within the generated SWF file at compile-time when Flex creates the SWF file for your application. Use this property with a container that is a child of a <code>ViewStack</code> container associated with a <code>LinkBar</code> or <code>TabBar</code> container, or with the children of a <code>TabNavigator</code> container. The icon appears in tab of the <code>TabBar</code> or <code>TabNavigator</code> , or in the link of the <code>LinkBar</code> corresponding to the container. The container scales the image to fit the tab size of a <code>TabBar</code> or <code>TabNavigator</code> container, or the link size of the <code>LinkBar</code> container. |
| <code>label</code> | String | When this container is a child of a navigator container, specifies the text that appears in the navigation control of the navigator container. For example, it specifies the text that appears in the tab of a <code>TabNavigator</code> or <code>TabBar</code> container, or the text in the link of a <code>LinkBar</code> container. |
| <code>numChildren</code> | Number | A read-only property containing the number of container children. |

| Property | Type | Description |
|------------------------------|--------|--|
| <code>vLineScrollSize</code> | Number | Specifies the number of pixels to move when the user selects the up or down arrow in a vertical scroll bar, or presses the Up or Down Arrow key. The default value is 5. |
| <code>vPageScrollSize</code> | Number | Specifies the number of pixels to move when the user clicks the scroll bar track of a vertical scroll bar. The default value is 20. |
| <code>vPosition</code> | Number | Specifies the pixel position of the slider in a vertical scroll bar, where a value of 0 corresponds to the upper end of the scroll bar. The default value is 0. |
| <code>vScrollPolicy</code> | String | Specifies when to include a vertical scroll bar: <ul style="list-style-type: none"> • <code>on</code> Causes the container to always include a scroll bar. • <code>off</code> Always excludes the scroll bar. • <code>auto</code> (default) Allows the container to include it only when the height of its children is larger than the height of the container. |

Using styles to configure a container

You use styles to set some container characteristics, such as margins and alignment. The list of styles available to each container includes those inherited from its parent and those defined for the container itself.

The following table describes some of the most common styles that you use with containers:

| Style | Type | Defining class | Description |
|------------------------------|--------|----------------|--|
| <code>backgroundAlpha</code> | Number | View | Adjusts the alpha level of the SWF file or image defined by <code>backgroundImage</code> , or the color defined by the <code>backgroundColor</code> property inherited from <code>UIComponent</code> . Valid values range from 0 to 100. The default value is 100. |
| <code>backgroundImage</code> | URL | View | <p>Specifies a URL to an external SWF file or JPEG, GIF, SVG, or PNG image file that appears in the background of the container. All children of the container appear on top of the image.</p> <p>You can use two formats for the syntax with this property:</p> <pre>backgroundImage="@Embed('re10rAbsoluteURL')"</pre> <p>The referenced image is packaged within the generated SWF file at compile-time when Flex creates the SWF file for your application.</p> <p>This form supports the import of GIF, PNG, JPEG, and SVG images, and SWF files.</p> <pre>backgroundImage="re10rAbsoluteURL"</pre> <p>Flex loads the referenced image file at runtime; it is not packaged as part of the generated SWF file.</p> <p>This form only supports the import of JPEG images and SWF files.</p> |

| Style | Type | Defining class | Description |
|------------------------------|--------|----------------------|---|
| <code>backgroundSize</code> | String | View | Scales the image specified by <code>backgroundImage</code> to different percentage sizes. By default, the value is <code>auto</code> , which maintains the original size of the image. A value of 100% stretches the image to fit the entire screen. You must include the percent sign with the value. |
| <code>borderStyle</code> | String | UIObject | Specifies the border style of the container. Possible values are <code>solid</code> , <code>none</code> , <code>inset</code> , and <code>outset</code> . The default value is <code>none</code> . |
| <code>marginLeft</code> | Number | UIObject | Specifies the number of pixels between the container's left border and its content area. |
| <code>marginRight</code> | Number | UIObject | Specifies the number of pixels between the container's right border and its content area. |
| <code>marginTop</code> | Number | Individual container | Specifies the number of pixels between the container's top border and its content area. |
| <code>marginBottom</code> | Number | Individual container | Specifies the number of pixels between the container's bottom border and its content area. |
| <code>horizontalAlign</code> | Number | Individual container | Specifies the horizontal alignment of children. Possible values are <code>left</code> , <code>center</code> , and <code>right</code> . The default value is <code>left</code> . |
| <code>verticalAlign</code> | Number | Individual container | Specifies the vertical alignment of children. Possible values are <code>top</code> , <code>middle</code> , and <code>bottom</code> . The default value is <code>top</code> . |
| <code>horizontalGap</code> | Number | Individual container | Specifies the number of pixels between children in the horizontal direction. |
| <code>verticalGap</code> | Number | Individual container | Specifies the number of pixels between children in the vertical direction. |

For more information on styles, see [Chapter 16, “Using Styles and Fonts,” on page 395](#).

Handling container events

All containers inherit the events defined by the `UIObject` and `UIComponent` classes. For more information on these classes, see [Chapter 1, “Using Flex Components,” on page 15](#).

In addition, they also inherit events from the `View` and `Container` classes. The following table describes the events inherited from the `View` and `Container` classes:

| Event | Description |
|-----------------------------|---|
| <code>childCreated</code> | Broadcast after a child has been created in a container. If a container has three children, it broadcasts three <code>childCreated</code> events, one after each child is created, and then broadcasts a <code>childrenCreated</code> event after all children have been created. The <code>relatedNode</code> property of the event object contains a copy of the created child object. |
| <code>childDestroyed</code> | Broadcast before a child of a container is destroyed. The <code>relatedNode</code> property of the event object contains a copy of the destroyed child object. |

| Event | Description |
|--------------------------------|---|
| <code>childIndexChanged</code> | <p>Broadcast after a container child's index has changed.</p> <p>The event object contains the following properties:</p> <ul style="list-style-type: none"> • <code>relatedNode</code> A copy of the moved child object. • <code>prevValue</code> The child's old index. • <code>newValue</code> The child's new index. |
| <code>childrenCreated</code> | <p>Broadcast after a container creates all of its children. This event lets you perform one-time initialization of a container's children, just after they get created.</p> <p>This event is useful with a container that is an immediate child of a navigator container, such as the <code>ViewStack</code>, <code>TabNavigator</code>, and <code>Accordion</code> navigator containers.</p> <p>For example, a <code>ViewStack</code> container creates all its immediate child containers, and all the children of its first visible child container. For the first visible child container, <code>childrenCreated</code> gets broadcast. Then, as the user moves to each additional child of the <code>ViewStack</code>, the event gets dispatched for that container.</p> <p>The event object contains the following properties:</p> <ul style="list-style-type: none"> • <code>type</code> A string set to <code>childrenCreated</code>. • <code>target</code> A reference to the child. |
| <code>scroll</code> | <p>Broadcast when the container is scrolled.</p> <p>The event object contains the following properties:</p> <ul style="list-style-type: none"> • <code>type</code> A string set to <code>scroll</code>. • <code>direction</code> The string <code>vertical</code> or <code>horizontal</code>, depending on the scroll direction. • <code>position</code> The new position of the scroll bar. • <code>delta</code> The number of pixels the scroll bar moved. |

Using behaviors with containers

The behaviors available for use with a container are those inherited by all components from the `UIObject` and `UIComponent` classes. For more information on behaviors, see [Chapter 18, “Using Behaviors,”](#) on page 453.

Using skins

The skins available for use with a container are described with the documentation of the individual containers.

CHAPTER 5

Using the Application Container

Macromedia Flex defines a default Application container that lets you start adding content to your application without having to explicitly define another container. The Application container provides support for an Alert pop-up dialog box that you can use to signal errors to users.

The Application container supports an application preloader that uses a progress bar to show the download progress of an application SWF file. You can override the default progress bar to define your own custom progress bar.

This chapter describes how to use an Application container.

Contents

| | |
|---|---------------------|
| Using the Application container. | 191 |
| Application container syntax | 195 |
| Showing the download progress of an application | 200 |

Using the Application container

Flex defines a default, or Application, container that lets you start adding content to your application without having to explicitly define another container. Flex creates this container from the `<mx:Application>` tag, the first tag in an MXML application file.

While you might find it convenient to use the Application container as the only container in your application, in most cases you will explicitly define at least one more container before you add any controls to your application. Often, you use a Panel container as the first container after the `<mx:Application>` tag.

An application container has the following default properties:

| Property | Default |
|--------------------|---|
| Default size | The size of the browser window. |
| Child alignment | Children are arranged in a vertical column. |
| Child sizing rules | Children with percentage values for <code>height</code> or <code>width</code> are resized in the corresponding direction. |

| Property | Default |
|----------------------------|---------------------------------------|
| Default margins | Top, bottom, left, right = 24 pixels. |
| Child horizontal alignment | Centered. |
| Background image | Gray gradient. |
| Background size | 100%. |

Sizing an Application container

An Application container arranges its children in a single vertical column. You can set the height and width of the Application container using explicit pixel values, or using percent values, where the percent values are relative to the size of the browser window. By default, the Application container has a height and width of 100%, which means that it takes up the entire browser window.

The following example sets the size of the Application container to one-half of the width and height of the browser window:

```
<?xml version="1.0"? >
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    height="50%" width="50%">
    ...
</mx:Application>
```

The advantage to using percentages to specify the size is that Flex can resize your application as the user resizes the browser window. Flex maintains the Application size as a percentage of the browser window as the user resizes it.

Note: You can use the `setSize()` method to set the height and width of the Application container. However, you cannot pass sizes to this method as percentages; you can only specify sizes in pixel values.

If you set the `width` and `height` properties of your components to percentage values, your components can also resize as your application resizes, as the following example shows:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    width="100%" height="100%">

    <mx:Panel title="Main Application" width="100%" height="100%">
        <mx:HDividedBox width="100%" height="100%">
            <mx:TextArea width="100%" height="50%" />
            <mx:VDividedBox width="100%" height="50%">
                <mx:DataGrid width="50%" height="100%" />
                <mx:TextArea width="50%" height="100%" />
            </mx:VDividedBox>
        </mx:HDividedBox>
    </mx:Panel>
</mx:Application>
```


The following example uses explicit pixel values to size the Application container:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    height="100" width="150">

    <mx:Panel title="Main Application" />
        <mx:TextInput id="mytext" text="Hello" />
        <mx:Button id="mybutton" label="Get Weather" />
    </mx:Panel>
</mx:Application>
```

If any children of the Application container are larger than the container's size, Flex adds scroll bars to the container. If you want to set a child container to fill the entire Application container, the easiest method is to set the `width` and `height` properties to 100%. If you use the values of the `width` and `height` properties of the Application container, you must subtract the margins, or your child container will be larger than the available space, which results in scroll bars. If you want your child container to fill the entire Application container, you must set all margins to 0.

In the following example, the VBox container expands to fill all of the available space:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" width="100"
    height="100">
    <mx:VBox width="100%" height="100%" backgroundColor="#A9C0E7">
    </mx:VBox>
</mx:Application>
```

In the following example, the VBox container is larger than the available space within the application container, which results in scroll bars:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" width="100"
    height="100">
    <mx:VBox width="100" height="100" backgroundColor="#A9C0E7">
    </mx:VBox>
</mx:Application>
```

In the following example, the Application container has no margins, which lets its child VBox container fill the entire window:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" width="100"
    height="100" marginTop="0" marginBottom="0" marginLeft="0" marginRight="0">
    <mx:VBox width="100%" height="100%" backgroundColor="#A9C0E7">
    </mx:VBox>
</mx:Application>
```

Overriding the default Application container styles

By default, the Application container has default style properties that define the following visual aspects of a Flex application:

| Property | Default |
|------------------------------|---|
| <code>backgroundImage</code> | Gray gradient |
| <code>backgroundSize</code> | 100%. When you set this property at 100%, the background image takes up the entire Application container. |

| Property | Default |
|-----------------|-----------|
| horizontalAlign | Centered |
| marginTop | 24 pixels |
| marginLeft | 24 pixels |
| marginBottom | 24 pixels |
| marginRight | 24 pixels |

You can override these default values in your application to define your own default style properties. For example, to set all margins to 0 pixels, remove the default background image, and left align the children, you can specify the `plain` style, as the following example shows:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    styleName="plain" >
```

This style setting does not change the default `backgroundColor`, which means that you still see the gray background during application loading. You can set the `backgroundColor` style property to remove the gray background, as the following example show:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    styleName="plain" backgroundColor="0xFFFFFF" />
```

You can also use the `<mx:Style>` tag in your application to specify alternative style values, as the following example shows:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

    <!-- Style definition for the entire application. -->
    <mx:Style>
        Application {
            marginLeft: 10px;
            marginRight: 10px;
            marginTop: 10px;
            marginBottom: 10px;
            horizontalAlign: "left";
            backgroundImage: " ";
            backgroundSize: " ";
        }
    </mx:Style>

    <mx:Panel title="Main Application" />

        <mx:TextInput id="mytext" text="Hello" />
        <mx:Button id="mybutton" label="Get Weather" click="getTemp(myzip);" />
    </mx:Panel>
</mx:Application>
```

This example removes the background image, sets all margins to 10 pixels, and left aligns children. For more information on using styles, see [Chapter 16, “Using Styles and Fonts,” on page 395](#).

Application container syntax

You use the `<mx:Application>` tag to define Application containers. This tag inherits all the properties of the classes `MovieClip`, `UIObject`, `UIComponent`, `View`, `Container`, and `Box`, except for the `direction` property of the `Box` class. For a list of these properties, see [“Configuring containers” on page 185](#).

This container also defines the following properties and methods:

| Property/Methods | Type | Use | Description |
|-------------------------------|---------|----------|---|
| <code>application</code> | Object | Property | Contains a reference to the Application object. Use this property when you want to refer to the top-level application from a script of another application more than one level down in the hierarchy. The alternative would be to use multiple <code>parentApplication</code> references. |
| <code>resetHistory</code> | Boolean | Property | When <code>true</code> , specifies that the history state of the application is reset to its initial state whenever the application is reloaded. The default value is <code>true</code> . Applications are reloaded when the following happen: <ul style="list-style-type: none">• The user selects the Refresh button in the browser.• The user navigates to another web page, and then selects the Back button in the browser to return to the Flex application.• The user loads a Flex application from the Favorites/Bookmarks menu in the browser. For more information on the history manager, see Chapter 23, “Using the History Manager,” on page 569 . |
| <code>alert()</code> | | Method | Creates an Alert box. For more information, see “Using alerts with the Application container” on page 197 . |
| <code>getURL()</code> | | Method | Loads a document from the specified URL into the specified window. If you reference this method from a file that does not contain the <code><mx:Application></code> tag, you must either import <code>mx.core.Application</code> , or use the fully qualified method name of <code>mx.core.Application.getURL()</code> . |
| <code>isFontEmbedded()</code> | | Method | Returns <code>true</code> if the specified font name is embedded. For more information, see Chapter 16, “Using Styles and Fonts,” on page 395 . |
| <code>popupWindow()</code> | | Method | Creates a pop-up <code>TitleWindow</code> container. For more information, see Chapter 7, “Using Navigator Containers,” on page 247 . |

Specifying options to the Application container

You can specify several options to the `<mx:Application>` tag to control your application. These options are not defined by the `Application` class; they are options built into the compiler and apply to any tag used as the root tag of your application.

The following table describes these options:

| Option | Type | Description |
|-----------------------------------|---------|---|
| <code>frameRate</code> | Number | Specifies the frame rate of the application. The default value is 24. |
| <code>pageTitle</code> | String | Specifies a string that appears in the title bar of the browser. This property provides the same functionality as the HTML <code><title></code> tag. |
| <code>lib</code> | String | Specifies one or more SWC files or SWS files that this application links to statically at compile time. This lets you specify resources at the application level that you would otherwise have to specify at the global level in the <code>flex-config.xml</code> file. |
| <code>preloader</code> | Path | Specifies the path of a SWC component class or ActionScript component class that defines a custom progress bar. A SWC component must be in the same directory as the MXML file or in the <code>WEB-INF/flex/user_classes</code> directory of your Flex web application. For more information, see “Showing the download progress of an application” on page 200 . |
| <code>rsl</code> | String | Specifies one or more SWC files or SWS files that this application uses as runtime shared libraries (RSLs). For more information on RSLs, see Chapter 25, “Using Runtime Shared Libraries,” on page 595 . |
| <code>scriptRecursionLimit</code> | Number | Specifies the maximum depth of the Macromedia Flash Player call stack before Flash Player stops. This is essentially the stack overflow limit. The default value is 1000. |
| <code>scriptTimeLimit</code> | Number | Specifies the maximum duration, in seconds, that an ActionScript event handler can execute before the Flash Player assumes that it is hung, and aborts it. The default value is 60 seconds. |
| <code>usePreloader</code> | Boolean | Specifies whether to disable the application preloader, <code>false</code> , or not, <code>true</code> . The default value is <code>true</code> . For more information, see “Showing the download progress of an application” on page 200 . |
| <code>theme</code> | File | Specifies a theme SWC file that contains skins for your application. Flex searches for the specified SWC file in a location relative to the MXML file. For more information, see Chapter 16, “Using Styles and Fonts,” on page 395 . |
| <code>xmlns</code> | URI | Specifies a namespace definition. For more information, see “Using XML namespaces” in Chapter 2, “Using MXML” in <i>Getting Started with Flex</i> . |

Using alerts with the Application container

The Application container provides support for an Alert control pop-up dialog box through its `alert()` method. All Flex components can call the `alert()` method to cause a pop-up modal dialog box to appear with a message and an optional title, buttons, and icons. The following figure shows an example of an Alert control pop-up dialog box:



The `alert()` method has the following syntax:

```
alert(message:String, title:String, flags:Number, clickHandler,  
      defaultButton:Number, icon:String) : Number
```

This method returns 0 if the Alert control is displayed, and nonzero otherwise.

The following table describes the arguments of the `alert()` method:

| Argument | Description |
|---------------|---|
| message | (Required) Specifies the text message displayed in the dialog box. |
| title | Specifies the dialog box title. If omitted, displays a blank title bar. |
| flags | Specifies the button(s) to display in the dialog box. The options are: <code>mx.controls.Alert.OK</code> OK button <code>mx.controls.Alert.CANCEL</code> Cancel button <code>mx.controls.Alert.YES</code> Yes button <code>mx.controls.Alert.NO</code> No button Each option is a bit value and can be combined with other options using the pipe ' ' operator. The default value is <code>mx.controls.Alert.OK</code> . |
| clickHandler | Specifies the handler for click events from the buttons. In addition to the standard click event parameters, the event object contains the <code>detail</code> field, which is set to the button flag that was clicked (<code>mx.controls.Alert.OK</code> , <code>mx.controls.Alert.CANCEL</code> , <code>mx.controls.Alert.YES</code> , or <code>mx.controls.Alert.NO</code>). |
| defaultButton | Specifies the default button using one of the possible values for the <code>flags</code> argument. This is the button that is selected when the user presses the Enter key. |
| icon | Specifies an icon to display to the left of the message text in the dialog box. |

You use the `alert()` method in your application, as the following example shows:

```
<?xml version="1.0"?>  
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >  
  
    <mx:Panel title="My Application">  
        <!-- Input TextInput control. -->  
        <mx:TextInput id="myInput" width="150" text="" />  
  
        <!-- Button control to trigger the copy. -->
```

```

        <mx:Button id="myButton" label="Copy Text"
            click='myText.text = myInput.text;alert("Text Copied!", "Alert Box",
                mx.controls.Alert.OK);' />

        <!-- Output TextArea control. -->
        <mx:TextInput id="myText" />
    </mx:Panel>
</mx:Application>

```

In this example, selecting the Button control copies text from the TextInput control to the TextArea control, and displays the Alert control.

You can also define an event handler for the Button control, as the following example shows:

```

<mx:Script>
    <![CDATA[

        function alertHandler(event)
        {
            myText.text = myInput.text;
            alert("Text Copied!", "Alert Box", mx.controls.Alert.OK);
        }
    ]]>
</mx:Script>

<mx:Button id="myButton" label="Copy Text" click="alertHandler(event)" />

```

If you want to create an Alert control pop-up dialog box from within a custom component or other file that does not contain the `<mx:Application>` tag, you can call either `mx.core.Application.alert()` or `mx.controls.Alert.show()` with the appropriate arguments. For more information on scoping, see [Chapter 12, “Working with ActionScript in Flex,”](#) on page 319.

Note: After the `alert()` method creates the dialog box, Flex continues processing of your application; it does not wait for the user to close the dialog box.

Using event handlers with the Alert control pop-up dialog box

The next example adds an event handler to the Alert control pop-up dialog box. An event handler lets you perform processing when the user selects a button of the Alert control. In this example, you only copy the text when the user selects the OK button in the Alert control:

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

    <mx:Script>
        <![CDATA[

            // Import Delegate.
            import mx.utils.Delegate;

            function alertHandler(event)
            {
                if (event.detail==mx.controls.Alert.OK)
                { myText.text = myInput.text; }
            }
        ]]>
    </mx:Script>

```

```

    }
  ]]>
</mx:Script>

<mx:Panel title="My Application">
  <!-- Input Textinput control. -->
  <mx:TextInput id="myInput" width="150" text="" />

  <!-- Output TextArea control. -->
  <mx:TextArea id="myText" />

  <!-- Button control to trigger the copy. -->
  <mx:Button id="myButton" label="Copy Text"
    click='alert("Copy Text?", "Alert",
      mx.controls.Alert.OK | mx.controls.Alert.CANCEL,
      Delegate.create(this, this.alertHandler),
      mx.controls.Alert.OK );'
  />
</mx:Panel>
</mx:Application>

```

This example uses the `Delegate` class to register the event handler. Event handlers execute in the scope of their event dispatcher. In this example, the event dispatcher is the `Button` control. However, in this example you want the event handler to run in the document scope so that it can access the `TextInput` and `TextArea` controls.

The `Delegate` class lets you register the event handler and specify the scope in which it executes. In the previous example, you use the `Delegate` class to specify a scope of `this`. The `this` scope corresponds to the document scope, which contains the `TextInput` and `TextArea` controls. For more information on scoping, see [Chapter 12, “Working with ActionScript in Flex,” on page 319](#).

Specifying an Alert control icon

You can include an icon in the `Alert` control that appears to the left of the `Alert` control text. This example modifies the example from the previous section to add the `Embed` metadata tag to import the icon. For more information on importing resources, see [Chapter 12, “Working with ActionScript in Flex,” on page 319](#).

```

<mx:Script>
  <![CDATA[

    [Embed(source="alertIcon.jpg")]
    var iconSymbol:String;

    function alertHandler(event)
    {
      if (event.detail==mx.controls.Alert.OK)
      { myText.text = myInput.text; }
    }
  ]]>
</mx:Script>

<mx:Panel title="My Application">

```

```

<!-- Input Textinput control. -->
<mx:TextInput id="myInput" width="150" text="" />

<!-- Output TextArea control. -->
<mx:TextArea id="myText" />

<!-- Button control to trigger the copy. -->
<mx:Button id="myButton" label="Copy Text"
    click='this.alert("Copy Text?", "Alert",
        mx.controls.Alert.OK | mx.controls.Alert.CANCEL,
        Delegate.create(this, this.alertHandler),
        mx.controls.Alert.OK,
        iconSymbol );'
/>
</mx:Panel>

```

Alert control styles

The Alert control supports the following styles:

| Style | Type | Use | Description |
|--------------|--------|-------|--|
| cornerRadius | Number | Style | Specifies the radius of the corners of the window frame. The default value is 8 pixels. |
| headerHeight | Number | Style | Specifies the height of the header. The default value is 28 pixels. |
| headerColors | Array | Style | Specifies an array of two colors for the header. The first element of the array specifies the top color; the second element specifies the bottom color. The default value is [0xE1E5EB, 0xF4F5F7]. |
| footerColors | Array | Style | Specifies an array of two colors for the footer. The first element of the array specifies the top color; the second element specifies the bottom color. The default value is [0xF4F5F7, 0xE1E5EB]. |

Showing the download progress of an application

The Application container supports an application preloader that uses a progress bar to show the download progress of an application SWF file. By default, the application preloader is enabled. The preloader keeps track of how many bytes have been downloaded and continually updates the progress bar.

The preloader appears during the application initialization period. The Application.creationComplete event dismisses the preloader.

The following figure shows the application preloader:



Disabling the application preloader

To disable the application preloader, you set the `usePreloader` property of the `Application` container to `false`, as the following example shows:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    usePreloader="false">
```

Using a custom progress bar

By default, the application preloader uses the `DownloadProgressBar` class in the `mx.preloaders` package to display download progress. To display a custom progress bar, you can create your own component that implements the `DownloadProgressBar` API. A custom progress bar component must extend the `MovieClip` class and should not use any of the standard Flex components, which would cause it to load too slowly to be an effective progress bar. You can implement a progress bar component as a SWC component or an `ActionScript` component. Do not implement a progress bar as an `MXML` component because it would load too slowly.

To use a custom progress bar, you set the `preloader` property of the `Application` container to the path of a SWC component class or `ActionScript` component class. A SWC component must be in the same directory as the `MXML` file or in the `WEB-INF/flex/user_classes` directory of your Flex web application. An `ActionScript` component can be in one of those directories or in a subdirectory of one of those directories. When a class is in a subdirectory, you specify the subdirectory location as the package name in the `preloader` value; otherwise, you specify the class name. The code in the following example specifies a custom progress bar called `CustomBar` that is located in the `WEB-INF/flex/user_classes/mycomponents/mybars` directory:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    preloader="mycomponents.mybars.CustomBar">
```

The following table describes the `DownloadProgressBar` API:

| Property/Method | Type | Use | Description |
|----------------------------|---------|---------------------|---|
| <code>indeterminate</code> | Boolean | Read-write property | The Preloader sets this property to <code>true</code> when it enters the application initialization state. A custom progress bar can alter its behavior at this point, or it can choose to ignore this request. No <code>setProgress()</code> calls are made while <code>indeterminate</code> is <code>true</code> . The default value is <code>false</code> . |
| <code>label</code> | String | Read-write property | The text to display when the progress bar is active. The preloader sets this value before displaying the progress bar. The default value is <code>""</code> . Implementing this property is optional. For example: <pre>function set label (val : String) { // Show the label in a text input myTextField.text = val; } function get label () : String { return myTextField.text;}</pre> |

| Property/Method | Type | Use | Description |
|-----------------|---------|---------------------|--|
| visible | Boolean | Read-write property | <p>Indicates whether the progress bar is visible. The default value is <code>false</code>.</p> <p>When the preloader determines that the progress bar should be displayed, it sets this value to <code>true</code>. When the preloader determines the progress bar should be hidden, it sets this value to <code>false</code>.</p> <p>Example:</p> <pre>function set visible (val : Boolean) { _alpha = 75; // Make the progress bar slightly transparent _visible = val; // Show the progress bar } function get visible () : Boolean { return _visible; }</pre> |
| setProgress() | | Method | <p>Updates the display of the progress bar with the current download information. A typical implementation divides the loaded value with the total value and displays a percentage.</p> <p>If you do not implement this method, you should create a progress bar that displays an animation to indicate to the user that a download is occurring.</p> <p>This method has the following signature:</p> <pre>setProgress(loaded, total)</pre> <ul style="list-style-type: none"> • loaded (required) The number of bytes of the application SWF file that have been downloaded. Must be of type <code>Number</code>. • total (required) The size of the application SWF file in bytes. Must be of type <code>Number</code>. <p>Example:</p> <pre>function setProgress(loaded:Number, total:Number) { if (loaded >=0 && total > 0) { var percent:Number = Math.ceil(100 * loaded / total); myPercentDisplay.text = percent + " %"; // Display the percentage as text myProgressBar.gotoandStop(percent); // Move the progress bar movie } }</pre> |

The following example shows the ActionScript code for a simple progress bar class:

```
class bar.TestBar extends MovieClip
{
    static var symbolName:String = "bar.TestBar";
    static var symbolOwner = bar.TestBar;
    var className:String = "TestBar";

    var progressText : TextField;
    var debugText : TextField;
    var __label : String = "";
    var __indeterminate : Boolean = false;
```

```

function TestBar() {
    trace("Change SWC TestBar constructor progressText = " + progressText + "
h = " + _height);
    _visible = true;
    display("Starting TestBar");
    // _visible = false;
}

function display(val : String) {
    trace("Calling display debugText.text = " + debugText.text);
    debugText.text += "\n" + val;
}

function set label(val:String) : Void {
    __label = val;
    display("Set Label to " + val);
}

function get label():String {
    return __label;
}

function setProgress(loaded:Number,total:Number) {
    progressText.text = "LOADED: " + loaded + " TOTAL: " + total;
}

function set indeterminate(val:Boolean) {
    __indeterminate = val;
    if (val)
        progressText.text = "Initializing . . . ";
}

function get indeterminate() : Boolean {
    return __indeterminate;
}

function set visible(val:Boolean) {
    _visible = val;
}

function get visible() : Boolean {
    return _visible;
}
}

```

For more information about creating ActionScript components, see [Chapter 14, “Creating ActionScript Components,”](#) on page 359.

CHAPTER 6

Using Layout Containers

Layout containers define rectangular regions of Macromedia Flash Player drawing surface, and provide a hierarchical structure to arrange and configure the components, such as Button and ComboBox controls, of a Macromedia Flex application. To use a layout container, you first create the container in MXML, then add the components that define your application.

This chapter describes layout containers, layout usage, and syntax. This chapter also includes descriptions of all Flex layout containers, and examples.

Contents

| | |
|--|-----|
| About layout containers | 206 |
| Canvas layout container | 206 |
| Box layout container | 209 |
| ControlBar layout container | 211 |
| DividedBox layout container | 212 |
| Form layout container | 215 |
| Grid layout container | 229 |
| Panel layout container | 234 |
| Tile layout container | 237 |
| TitleWindow layout container | 239 |

About layout containers

A layout container controls the sizing and positioning of the child controls and child containers defined within it. For example, a Form layout container sizes and positions its children in a layout similar to an HTML form.

Flex provides the following layout containers:

- Canvas
- Box
- ControlBar
- DividedBox
- Tile
- Grid
- Form
- Panel
- TitleWindow

The following sections describe how to use each of the Flex layout containers.

Canvas layout container

A Canvas layout container defines a rectangular region in which you place child containers and controls. It is the only container that lets you explicitly specify the location of its children within the container. That is, you are responsible for using the `x` and `y` properties of each child to specify its location in a Canvas layout container.

Flex does not resize any children of a Canvas layout container by default. Therefore, if you want to change the size of a child from its preferred size, use the `height` and `width` properties of the child to set it to a fixed size or to a percentage of the Canvas container's size. Flexible components retain their position in the container. That is, their upper left corner is fixed to the specified `x` and `y` coordinates. Only space below and to the right of this position is available to the component.

In the following example, a VBox container is set to fill 100% of its parent Canvas container. The space above and to the left of its coordinates remains empty:

```
<mx:Canvas width="100" height="100" borderStyle="solid">
  <mx:VBox width="100%" height="100%" x="20" y="20"
    backgroundColor="#A9C0E7">
  </mx:VBox>
</mx:Canvas>
```

This example produces the following image:



If you use percentage sizing, be aware that some of your components may overlap, since resizing the Canvas container does not change the coordinates of the components, just their size. In addition, if you use borders on your Canvas containers, the children components may overlap the borders, because the Canvas container does not adjust the coordinate system to account for the thickness of borders.

In the following example, the size and position of each component is carefully calculated to ensure that none of the components overlap:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
  width="100" height="100" >
  <mx:Canvas id="chboard" >
    <mx:Image source="BlackBox.jpg" width="10" height="10" x="0" y="0" />
    <mx:Image source="BlackBox.jpg" width="10" height="10" x="20" y="0" />
    <mx:Image source="BlackBox.jpg" width="10" height="10" x="40" y="0" />
    <mx:Image source="BlackBox.jpg" width="10" height="10" x="10" y="10" />
    <mx:Image source="BlackBox.jpg" width="10" height="10" x="30" y="10" />
    <mx:Image source="BlackBox.jpg" width="10" height="10" x="0" y="20" />
    <mx:Image source="BlackBox.jpg" width="10" height="10" x="20" y="20" />
    <mx:Image source="BlackBox.jpg" width="10" height="10" x="40" y="20" />
    <mx:Image source="BlackBox.jpg" width="10" height="10" x="10" y="30" />
    <mx:Image source="BlackBox.jpg" width="10" height="10" x="30" y="30" />
    <mx:Image source="BlackBox.jpg" width="10" height="10" x="0" y="40" />
    <mx:Image source="BlackBox.jpg" width="10" height="10" x="20" y="40" />
    <mx:Image source="BlackBox.jpg" width="10" height="10" x="40" y="40" />
  </mx:Canvas>
</mx:Application>
```

This example produces the following image:



If you set the `width` and `height` properties of one of the images to 20% but don't change the positions accordingly, that image overlaps other images in the checkerboard:



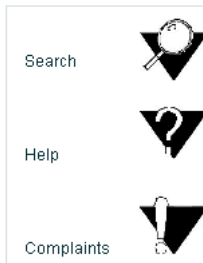
A Canvas container has the following default properties:

| Property | Default |
|-----------------|--|
| Preferred size | Large enough to hold all of its children at the preferred sizes of the children. |
| Default margins | Top, bottom, left, right = 0 pixels |

Canvas container example

You use the `x` and `y` properties of each child to specify the child's location in the Canvas container. These properties specify the `x` and `y` coordinates of a child relative to the upper-left corner of the Canvas container, where the upper-left corner is at coordinates (0,0). Values for the `x` and `y` coordinates can be positive or negative integers. You can use negative values to place a child outside the visible area of the container, and then use `ActionScript` to move the child to the visible area, possibly as a response to an event.

The following figure shows a Canvas container with three Link controls and three Image controls:



The following MXML code creates this Canvas container:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">

    <mx:Canvas id="myCanvas" borderStyle="solid" backgroundColor="white" >
        <mx:Link label="Search" click="getURL('http://mycomp.com/search')"
            x="10" y="30"/>
        <mx:Image source="search.jpg" height="50" width="50" x="100" y="10"/>

        <mx:Link label="Help" click="getURL('http://mycomp.com/help')"
            x="10" y="100"/>
        <mx:Image source="help.jpg" height="50" width="50" x="100" y="75" />

        <mx:Link label="Complaints" click="getURL('http://mycomp.com/complain')"
            x="10" y="170"/>
        <mx:Image source="complaint.jpg" height="50" width="50" x="100" y="140" />

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

You can build logic into your application to reposition a child of a Canvas container at runtime. For example, the following code repositions an input text box with the id of `text1` to `x=110`, `y=110` in response to a button click:

```
<mx:TextInput id="text1" text="Move me" x="50" y="50" />
<mx:Button id="button1" label="Move text1" x="50" y="300"
    click="text1.x=110; text1.y=110;" />
```


Canvas container syntax

You use the `<mx:Canvas>` tag to define a Canvas layout container in MXML. The Canvas layout container inherits all the properties of its parent classes: `MovieClip`, `UIObject`, `UIComponent`, `View`, and `Container`, but adds no new ones. For a list of these properties, see [“Configuring containers” on page 185](#).

Note: The Canvas container inherits the `marginRight` and `marginLeft` properties, but ignores them.

Box layout container

A Box layout container lays out its children in a single vertical column or a single horizontal row. You use the `direction` property of a Box container to determine either vertical (default) or horizontal layout.

A Box container has the following default properties:

| Property | Default |
|-----------------|---|
| Direction | Vertical |
| Preferred size | Vertical Box Height is large enough to hold all of its children at the preferred height of the children, plus any vertical gap between the children, plus the top and bottom container margins. Width is the preferred width of the widest child, plus the left and right container margins. Horizontal Box Height is the preferred height of the tallest child, plus the top and bottom container margins. Width is large enough to hold all of its children at the preferred width of the children, plus any horizontal gap between the children, plus the left and right container margins. |
| Default margins | Top, bottom, left, right = 0 pixels. |

Box layout container example

The following figure shows a Box container with a horizontal layout and one with a vertical layout:



Box container with horizontal layout



Box container with vertical layout (default)

Note: To lay out children in multiple rows or columns, use a `Tile` or `Grid` container. For more information, see [“Tile layout container” on page 237](#) and [“Grid layout container” on page 229](#).

The following example creates a Box container with a vertical layout:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

    <mx:Box direction="vertical" borderStyle="solid" marginTop="10"
        marginBottom="10" marginLeft="10" marginRight="10" >
        <mx:Button id="fname" label="Button 1" />
        <mx:Button id="lname" label="Button 2" />
        <mx:Button id="addr1" label="Button 3" />
        <mx:ComboBox id="state" />
    </mx:Box>
</mx:Application>
```

Typically, you use the VBox (vertical box) and HBox (horizontal box) containers as shortcuts so you do not have to specify the `direction` property. The following code example is equivalent to the previous example, except that this example defines a vertical Box container using the

```
<mx:VBox> tag:

<mx:VBox borderStyle="solid" marginTop="10" marginBottom="10"
    marginLeft="10" marginRight="10" >
    <mx:Button id="fname" label="Button 1" />
    <mx:Button id="lname" label="Button 2" />
    <mx:Button id="addr1" label="Button 3" />
    <mx:ComboBox id="state" />
</mx:VBox>
```

Sizing a Box container

If none of the Box container's children are resizable, an HBox container's `minWidth` property is set to be the sum of the children's widths, plus space for gaps and margins. The HBox container's `minHeight` property is set to be the largest of the children's heights, plus space for margins. Similarly, a VBox container's `minHeight` property is set to be the sum of the children's heights, plus space for gaps and margins. The VBox container's `minWidth` property is set to be the largest of the children's widths, plus space for margins.

If all of the Box container's children are resizable, an HBox container's `minWidth` property is the sum of the children's minimum widths, plus space for gaps and margins. The HBox container's `minHeight` property is the largest of the children's minimum heights, plus space for margins. Similarly, a VBox container's `minHeight` property is the sum of the children's minimum heights, plus space for gaps and margins. The VBox container's `minWidth` property is the largest of the children's minimum widths, plus space for margins.

If the Box container has a mixture of resizable and fixed-size children, the `minWidth` and `minHeight` properties are calculated using a combination of these two rules.

Box container syntax

You use the `<mx:Box>`, `<mx:VBox>`, and `<mx:HBox>` tags to define Box containers. These tags inherit all the properties of the classes `MovieClip`, `UIObject`, `UIComponent`, `View`, and `Container`. For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*.

These containers also define the following optional property:

| Property | Type | Use | Description |
|-----------|--------|----------|--|
| direction | String | Property | For a Box container only, specifies the orientation of the Box container. Possible values are <code>horizontal</code> and <code>vertical</code> . The default value is <code>vertical</code> . |

ControlBar layout container

You use the ControlBar container with a Panel or TitleWindow container to hold components that can be shared by the other children in the Panel or TitleWindow container. For a product catalog, the ControlBar container can hold the Flex controls to specify quantity and to add an item to a shopping cart, as the following figure shows:



The ControlBar class is a subclass of the HBox class and, therefore, inherits the layout characteristics of the HBox container. The ControlBar container has the following default properties:

| Property | Default |
|-----------------|---|
| Direction | Horizontal |
| Preferred size | Height is the preferred height of the tallest child, plus the top and bottom container margins. Width is large enough to hold all of its children at the preferred width of the children, plus any horizontal gap between the children, plus the left and right container margins. |
| Default margins | Top, bottom, left, right = 10 pixels. |

Creating a ControlBar container

You use the `<mx:ControlBar>` tag to define a ControlBar control in MXML. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block. You specify the `<mx:ControlBar>` tag as the last child tag of an `<mx:Panel>` tag, as the following example shows:

```
<mx:Panel title="My Application" marginTop="10" marginBottom="10"
  marginLeft="10" marginRight="10" >
```

```

<mx:HBox width="250">
  <!-- Area for your catalog. -->
</mx:HBox>

<mx:ControlBar width="250">
  <mx:Label text="Quantity" />
  <mx:NumericStepper />
  <!-- Use Spacer to push Button control to the right. -->
  <mx:Spacer width="100%" />
  <mx:Button label="Add to Cart" click="addToCart()" />
</mx:ControlBar>
</mx:Panel>

```

ControlBar container syntax

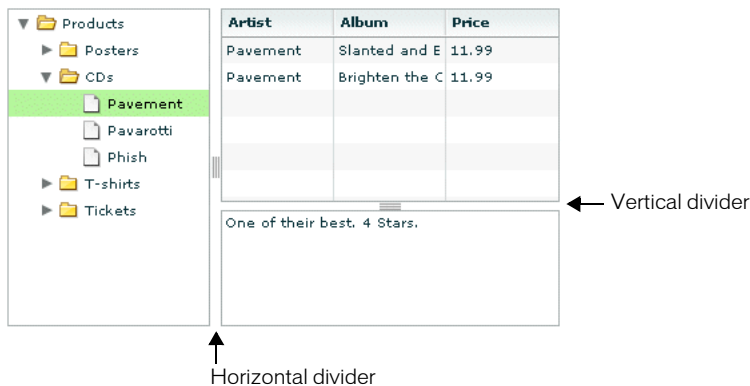
You use the `<mx:ControlBar>` tag to define a `ControlBar` container. The `ControlBar` container inherits all the properties of the `HBox` class, but adds no new ones.

Note: While the `ControlBar` container inherits the `borderStyle`, `backgroundColor`, and `backgroundImage` styles from the `HBox` container, you cannot set them.

DividedBox layout container

The `DividedBox` layout container lays out its children horizontally or vertically, much like a `Box` container, except that it inserts a divider between each child. Users can use a mouse to move the dividers to resize the area of the container allocated to each child. You use the `direction` property of a `DividedBox` container to determine vertical (default) or horizontal layout.

The following figure shows a `DividedBox` container:



In this figure, the outermost container is a horizontal `DividedBox` container. The horizontal divider marks the border between a `Tree` control and a vertical `DividedBox` container.

The vertical `DividedBox` container holds a `DataGrid` control (top) and a `TextArea` control (bottom). The vertical divider marks the border between these two controls.

A DividedBox container has the following default properties:

| Property | Default |
|-----------------|---|
| Direction | Vertical |
| Preferred size | Vertical DividedBox Height is large enough to hold all of its children at the preferred height of the children, plus any vertical gap between the children, plus the top and bottom container margins. Width is the preferred width of the widest child, plus the left and right container margins. Horizontal DividedBox Height is the preferred height of the tallest child, plus the top and bottom container margins. Width is large enough to hold all of its children at the preferred width of the children, plus any horizontal gap between the children, plus the left and right container margins. |
| Default margins | Top, bottom, left, right = 0 pixels. |
| Default gap | Horizontal and vertical gaps are 10 pixels. |

Creating a DividedBox container

You define a DividedBox container in MXML using the `<mx:DividedBox>` tag, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The following example creates the DividedBox container shown in the figure in [“DividedBox layout container” on page 212](#):

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

    <mx:DividedBox direction="horizontal" width="100%" height="100%">
        <mx:Tree id="tree1" width="100%" height="100%" >
            ...
        </mx:Tree>

        <mx:DividedBox direction="vertical" width="100%" height="100%">
            <mx:DataGrid id="myGrid" width="50%" height="100%" />
            <mx:TextArea id="currentMessage" width="50%" height="100%" />
        </mx:DividedBox>
    </mx:DividedBox>

</mx:Application>
```

Typically, you use the `VDividedBox` (vertical DividedBox) and `HDividedBox` (horizontal DividedBox) containers as shortcuts so that you do not have to specify the `direction` property. The following code example is equivalent to the preceding example, except that this example defines a vertical DividedBox container using the `<mx:VDividedBox>` tag:

```
<mx:HDividedBox width="100%" height="100%">
    <mx:Tree id="tree1" width="100%" height="100%">
        ...
    </mx:Tree>
```

```

<mx:VDividedBox>
    <mx:DataGrid id="myGrid" width="50%" height="100%" />
    <mx:TextArea id="currentMessage" width="50%" height="100%" />
</mx:VDividedBox>
</mx:HDividedBox>

```

Using the dividers

The dividers of a `DividedBox` container let you resize the area of the container allocated for a child. However, for the dividers to function, the child has to be resizable. So, a child with an explicit height or width cannot be resized in the corresponding direction using a divider.

By default, Flex sizes the children of a `DividedBox` container to the child's preferred size. You can use the dividers to resize a child up to its maximum size, or down to its minimum size.

If you specify a percentage value for the `height` or `width` properties of a child to make it resizable, Flex initially sizes the child to the specified percentage, if possible, and then can resize the child to take up all available space using the rules described in [“Sizing components” on page 26](#). When you use the `DividedBox` container, you typically use percentage sizing for its children to make them resizable.

To constrain the minimum size or maximum size of an area of the `DividedBox`, set an explicit value for the `minWidth` and `minHeight` properties or the `maxWidth` and `maxHeight` properties of the children within that area.

Using live dragging

By default, the `DividedBox` container disables live dragging. This means that the `DividedBox` container does not update the layout of its children until the user finishes dragging the divider, as signaled when the user releases the mouse button on a selected divider.

You can configure the `DividedBox` container to use live dragging by setting the `liveDragging` property to `true`. With live dragging enabled, the `DividedBox` container updates its layout as the user moves a divider. In some cases, you may encounter decreased performance if you enable live dragging.

Using `DividedBox` events

The `DividedBox` container supports the events that it inherits from its parents, plus it defines the following additional events:

dividerPressed Broadcast when a user selects any divider.

dividerDragged Broadcast while the user drags the divider. This event is broadcast whenever the user moves the mouse while a divider is selected.

dividerReleased Broadcast when the user releases the divider, but before Flex resizes the container children.

DividedBox container syntax

You use the `<mx:DividedBox>`, `<mx:VDividedBox>`, and `<mx:HDividedBox>` tags to define DividedBox containers. These tags inherit the properties and methods of the classes `MovieClip`, `UIObject`, `UIComponent`, `View`, and `Container`. The DividedBox container also inherits the properties and methods of the `Box` container. For more information, see [“Box layout container” on page 209](#). For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*.

These containers also define the following optional properties:

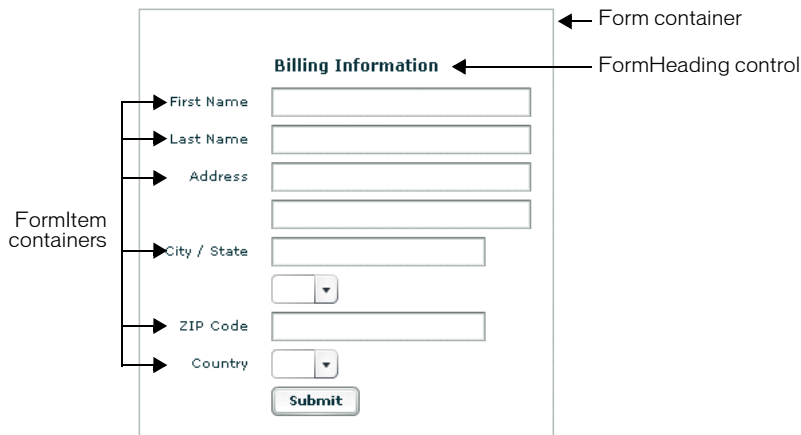
| Property | Type | Use | Description |
|------------------------------|---------|----------|---|
| <code>liveDragging</code> | Boolean | Property | Specifies whether to continuously resize children as a divider is dragged, <code>true</code> , or not, <code>false</code> . The default value is <code>false</code> . |
| <code>dividerDragged</code> | | Event | <ul style="list-style-type: none">Broadcast when a user drags a divider. |
| <code>dividerPressed</code> | | Event | <ul style="list-style-type: none">Broadcast when a divider is selected. |
| <code>dividerReleased</code> | | Event | <ul style="list-style-type: none">Broadcast after a divider has been released, but before Flex resizes the children. |

Form layout container

Forms are one of the most common methods web applications use to collect information from users. Forms are used for collecting registration, purchase, and billing information, and for many other types of data collection tasks.

Flex supports form development using the Form layout container and several child components of the Form container. The Form container lets you control the layout of a form, mark form fields as required or optional, handle error messages, and bind your form data to the Flex data model to perform data checking and validation. In addition, you can apply style sheets to configure the appearance of your forms.

You use three different components to create your forms, as the following figure shows:



The following table describes the types of components that you use to create forms in Flex:

| Container | Tag | Description |
|-------------|-------------------------------------|--|
| Form | <code><mx:Form></code> | Defines the container for the entire form. Use the <code>FormHeading</code> control and <code>FormItem</code> container to define content. You can also insert other types of components into a <code>Form</code> container. |
| FormHeading | <code><mx:FormHeading></code> | Defines a heading within your form. You can have multiple <code>FormHeading</code> controls within a single <code>Form</code> container. |
| FormItem | <code><mx:FormItem></code> | Defines one or more form children arranged horizontally or vertically. Children can be controls or other containers. A single <code>Form</code> container can hold multiple <code>FormItem</code> containers. |

Creating a Form container

The `Form` container is the outermost container of a Flex form. The primary use of the `Form` container is to control the sizing and layout of the contents of the form, including the size of labels and the gap between items.

The `Form` container always arranges its children vertically and left aligns them in the form. The following example shows the `Form` container definition for the form shown in the previous figure:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

    <mx:Form id="myForm" width="400" height="100" >

        <!-- Define Form Heading and FormItem components here -->

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

Creating a FormHeading control

A `FormHeading` control specifies an optional label for a group of `FormItem` containers. The left side of the label is aligned with the left side of the form.

The following example defines the `FormHeading` control for the example shown in the previous figure:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

    <mx:Form id="myForm" width="400" height="100" >

        <mx:FormHeading label="Billing Information" />

        <!-- Define FormItem containers here. -->

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

You can have multiple `FormHeading` controls in your form to designate multiple content areas. You can also use `FormHeading` controls with a blank `label` to create vertical space in your form.

Creating a FormItem container

A FormItem container specifies a single label and one or more child controls or containers. The label is vertically aligned with the first child in the FormItem container.

Form containers typically contain multiple FormItem containers, as the following figure shows:



In this example, you define three FormItem containers: one with the label First Name, one with the label Last Name, and one with the label Address. The Address FormItem container holds two controls to let a user enter two lines of address information. The other two FormItem containers each contain a single control.

When you create a FormItem container, you specify its direction as either vertical (default) or horizontal:

Vertical direction Flex positions children vertically to the right of the FormItem label.

Horizontal direction Flex positions children horizontally to the right of the FormItem label. If all children do not fit on a single row, they are divided into multiple rows with equal-sized columns. You can ensure all children fit on a single line by using flexible widths or by specifying explicit widths wide enough for all of the components.

The following example shows the FormItem container definitions for the example form:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

  <mx:Form id="myForm" width="400" >

    <mx:FormHeading label="Billing Information" />

    <mx:FormItem label="First Name" >
      <mx:TextInput id="fname" width="100%" />
    </mx:FormItem>

    <mx:FormItem label="Last Name" >
      <mx:TextInput id="lname" width="100%" />
    </mx:FormItem>

    <mx:FormItem label="Address" >
      <mx:TextInput id="addr1" width="100%" />
      <mx:TextInput id="addr2" width="100%" />
    </mx:FormItem>

    <mx:FormItem label="City / State" direction="horizontal">
      <mx:TextInput id="city" />
      <mx:ComboBox id="st" width="50%" />
    </mx:FormItem>

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

```

<mx:FormItem label="Zip Code" >
    <mx:TextInput id="zip" width="100" />
</mx:FormItem>

<mx:FormItem label="Country" >
    <mx:ComboBox id="cntry" />
</mx:FormItem>

<mx:FormItem>
    <mx:Button label="Submit"
        click="setValues(fnName.text, lname.text, addr1.text, addr2.text,
            city.text, st.value, zip.text, cntry.value);" />
</mx:FormItem>

</mx:Form>
</mx:Application>

```

Defining a default button

You use the `defaultButton` property of a container to define a default Button control. Pressing the Enter key while focus is on any form control activates the Button control just as if it was explicitly selected.

For example, a login form displays user name and password inputs and a submit Button control. Typically, the user types a user name, tabs to the password field, types the password, and presses the Enter key to submit the login information without explicitly selecting the Button control. To define this type of interaction, set the `defaultButton` to the `id` of the submit Button control, as the following example shows:

```

<mx:Script>
    <![CDATA[
        function submitLogin(event) {
            myWebService.Login.send();
        }
    ]]>
</mx:Script>

<mx:Form defaultButton="mySubmitButton" >
    <mx:FormItem label="Username">
        <mx:TextInput id="username" width="100"/>
    </mx:FormItem>
    <mx:FormItem label="Password">
        <mx:TextInput id="password" width="100" password="true"/>
    </mx:FormItem>
    <mx:FormItem>
        <mx:Button id="mySubmitButton" label="Login" click="submitLogin(event)"/>
    </mx:FormItem>
</mx:Form>

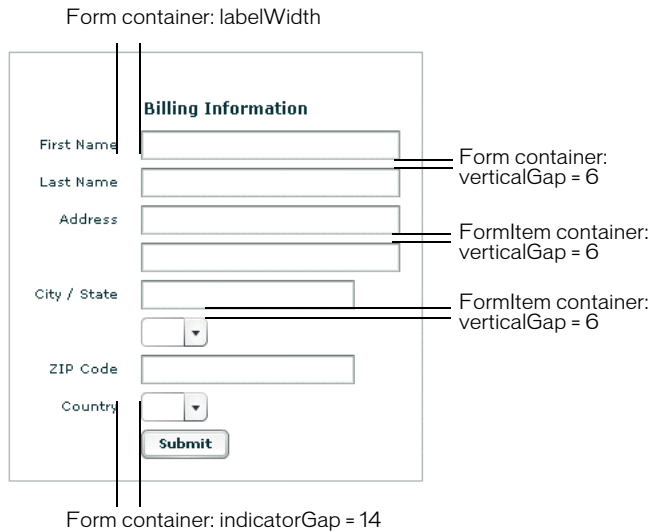
```

Note: The ComboBox control has a special meaning for the Enter key. When the drop-down list of a ComboBox control is open, pressing Enter selects the currently highlighted item in the ComboBox control; it does not activate the default button.

Aligning and spacing Form container children

All Form container labels are right-aligned, and all children are left-aligned in the container. You cannot override this alignment.

The following figure shows the spacing of Form container children that you can control:



The following table describes the parameters and default values that you use to control spacing:

| Component | Parameter | Description | Default |
|-------------|--------------|--|---|
| Form | verticalGap | Space between Form container children. | 6 pixels |
| | labelWidth | Width of labels. | Calculated by the container based on the child labels |
| | marginTop | Border spacing around children. | 16 pixels on all sides |
| | marginBottom | | |
| | marginLeft | | |
| | marginRight | | |
| FormHeading | indicatorGap | Gap between the end of the FormItem label and the child. | 14 pixels |
| | indicatorGap | Overrides the indicator gap set by the <code><mx:Form></code> tag. | 15 pixels |
| | verticalGap | Gap between the top of the component and the label text. | 16 pixels |

| Component | Parameter | Description | Default |
|-----------|--|--|---------------------------------|
| FormItem | direction | Direction of FormItem children: vertical or horizontal. | vertical |
| | horizontalGap | Horizontal spacing between children in a FormItem container. | 6 pixels |
| | labelWidth | The width for the FormItem heading. | The width of the label text |
| | marginTop marginBottom marginLeft marginRight | Border spacing around the FormItem. | 0 pixels on all sides |
| | verticalGap | Vertical spacing between children in a FormItem container. | 6 pixels |
| | indicatorGap | Overrides the indicator gap set by the <mx:Form> tag. | Determined by the <mx:Form> tag |

Sizing and positioning Form container children

The Form layout container arranges children in a vertical column. The area of the Form container that is designated for children does not encompass the entire Form container. Instead, it starts at the right of the area defined by any labels and the gap defined by the `indicatorGap` property. For example, if the width of the Form container is 500 pixels, and the labels and `indicatorGap` property allocate 100 pixels of that width, the width of the child area is 400 pixels.

By default, Flex stretches the Form layout children vertically to their preferred height. Flex then determines the preferred width of each child, and stretches the child's width to the next highest value of the child area's one-quarter, one-half, three-quarter, or full width.

For example, if a container has a child area 400 pixels wide, and the preferred width of a `TextArea` control is 125 pixels, Flex stretches the `TextArea` control horizontally to the next higher one-quarter, one-half, three-quarter, or full width of the child area. In this example, Flex stretches the `TextArea` to the 200-pixel boundary, which is one-half of the child area. This sizing algorithm only applies to components without an explicitly specified width, and prevents your containers from having ragged right edges caused by controls with different widths.

You can also explicitly set the height or width of any control in the form to either a pixel value or a percentage of the Form size by using the `height` and `width` properties of the child.

Defining required fields

Flex includes support for defining required input fields of a form. To define a required field, you specify the `required` property of the `FormItem` container. If specified, all the children of the `FormItem` container are marked as required.

Flex inserts a red asterisk (*) character as a separator between the `FormItem` label and the `FormItem` child to indicate a required field. For example, the following figure shows an optional ZIP code field and a required ZIP code field:



The image shows two text input fields. The first field is preceded by the text 'ZIP Code'. The second field is preceded by the text 'ZIP Code *', where the asterisk is red, indicating a required field.

The following example defines these fields:

```
<mx:FormItem label="ZIP Code" >
  <mx:TextInput id="zipOptional" width="100" />
</mx:FormItem>

<mx:FormItem label="ZIP Code" required="true">
  <mx:TextInput id="zipRequired" width="100" />
</mx:FormItem>
```

You can enable the required indicator of a `FormItem` child at runtime. This could be useful when the user input in one form field makes another field required. For example, you might have a form with a `CheckBox` control that the user selects to subscribe to a newsletter. Checking the box could make the user e-mail field required, as the following example shows:

```
<mx:FormItem label="Subscribe" >
  <mx:CheckBox label="Subscribe?" click="emAddr.required=true"/>
</mx:FormItem>

<mx:FormItem id="emAddr" label="e-mail address">
  <mx:TextInput id="emailAddr" />
</mx:FormItem>
```

Flex does not perform any automatic enforcement of a required field; it only marks fields as required. You must build your own validation logic into your form to enforce it. As part of your enforcement logic, you can use Flex validators. All of the validators supplied with Flex return an error if a field is empty; therefore, you can use them to display an error message for an empty Form control. For more information on using validators with forms, see [“Using a Flex data model to store form data” on page 223](#).

Storing and validating form data

As part of designing your form, you have to consider how you want to store your form data. In Flex, you have the following choices:

- Store the data within the form controls.
- Create a Flex data model to store your data.

One of the primary tasks in building robust and stable forms is the process of input error detection and data validation. Your decision on how you represent your data also affects how you perform data validation. As part of building your form, you can perform data validation using your own custom logic, take advantage of the Flex data validation mechanism, or use a combination of the two.

Using Form controls to hold your form data

The following example uses Form controls to store the form data:

```
<mx:Form id="myForm" >

    <mx:FormItem label="Zip Code" >
        <mx:TextInput id="zipCode"/>
    </mx:FormItem>
    <mx:FormItem label="Phone Number" >
        <mx:TextInput id="phoneNumber" />
    </mx:FormItem>

    <mx:FormItem>
        <mx:Button label="Submit"
            click="processValues(zipCode.text, phoneNumber.text);" />
    </mx:FormItem>

</mx:Form>

<mx:Script>
    <![CDATA[
        function processValues(zip, pn)
        {
            // Check to see if phoneNumber is a number.
            // Check to see if zipCode is less than 4 digits.
            // Process data.
        }
    ]]>
</mx:Script>
```

This example form defines two form controls: one for a ZIP code and one for a phone number. When you submit the form, you call a function that takes the two arguments that correspond to the data stored in each control. Your submit function can then perform any data validation on its inputs before processing the form data.

You don't have to pass the data to the submit function. The submit function can access the form control data directly, as the following example shows:

```
<mx:Script>
    <![CDATA[
        function processValues()
        {
            var inputZip:String = zipCode.text;
            var inputPhone:String = phoneNumber.text;
            // Check to see if pn is a number.
            // Check to see if zip is less than 4 digits.
            // Process data.
        }
    ]]>
</mx:Script>
```

The only problem with this technique is that the submit function is now specific to your form and cannot easily be used by other forms.

You typically validate user input before you submit the data to the server. When you store data in the form controls, you can either validate the user input within the submit function, or when a user enters data into the form.

To validate form data on user input, you write an event handler for the control for the `valueCommitted` event. The `valueCommitted` event is triggered when a user completes data entry into a control. The following example uses the `valueCommitted` event of a control to perform validation:

```
<mx:Script>
  <![CDATA[
    function validateFName()
    {
      // Perform validation.
      // On error, pop up an alert box.
      mx.controls.Alert.show("Please enter a valid first name string",
        "Alert Box",mx.controls.Alert.OK);
    }
  ]]>
</mx:Script>

<mx:FormItem label="First Name" >
  <mx:TextInput id="fname" valueCommitted="validateFName()"/>
</mx:FormItem>
```

If you validate the input data every time the user enters it, you might not have to do so again in your submit function. Or, you might still have to perform some validation in your submit function, especially if you want to make sure that two fields are valid when compared with each other.

For example, you can use event handlers to validate a ZIP code field and state field individually. But you might want to validate that the ZIP code is valid for the specified state before submitting the form data. To do so, you perform a second validation in the submit function.

Using a Flex data model to store form data

Flex provides a data validation mechanism as part of its data model. This mechanism lets you define the type of data contained within an input field, and generate errors when a user enters an incorrect value.

For example, you can define an input control to take a ZIP code. ZIP codes are five-digit or nine-digit numbers that contain no alphabetic characters. If a user attempts to enter an incorrect value, and you use the Flex data model with your form, Flex can automatically issue an error message.

To take advantage of the Flex validation mechanism, you have to define a data model for the form. A data model stores data in fields that represent each part of a specific data set. For example, a *person* model might store information such as a person's name, age, and phone number.

The following example defines a Flex data model that contains two values that correspond to the two input fields of a form:

```
<!-- Define data model. -->
<mx:Model id="myFormModel">
  <zipCodeModel>{zipCode.text}</zipCodeModel>
```

```

    <phoneNumberModel>{phoneNumber.text}</phoneNumberModel>
</mx:Model>

<!-- Define form. -->
<mx:Form borderStyle="solid" >

    <mx:FormItem label="Zip Code" >
        <mx:TextInput id="zipCodeModel"/>
    </mx:FormItem>

    <mx:FormItem label="Phone Number" >
        <mx:TextInput id="phoneNumberModel" />
    </mx:FormItem>

    <mx:FormItem>
        <mx:Button label="Submit"
            click="processValues();" />
    </mx:FormItem>
</mx:Form>

<mx:Script>
    <![CDATA[
        function processValues()
        {
            var inputZip:String = myFormModel.zipCodeModel;
            var inputPhone:String = myFormModel.phoneNumberModel;
            ...
            // Process data.
        }
    ]]>
</mx:Script>

```

You use the `<mx:Model>` tag to define the data model. Each child tag of the data model defines one field of the model. The tag body of each child tag in the model defines a *binding* to a form control. In this example, you bind the `zipCodeModel` model field to the text value of the `zipCode` `TextInput` control, and the `phoneNumberModel` field to the text value of the `phoneNumber` `TextInput` control. For more information on data models, see [Chapter 28, “Managing Data in Flex,” on page 629](#).

When you bind a control to a data model, Flex automatically copies data from the control to the model upon user input. In this example, your submit function accesses the data from the model, not directly from the form controls.

One of the big advantages of using a data model is that it lets you also use the Flex data validation mechanism. The validation mechanism performs automatic data validation and error reporting for your form data.

As part of the validation mechanism, Flex provides a set of data validators for the most common types of data collected by a form. You can use Flex validators with the following types of data:

- Credit card information
- Dates
- E-mail addresses

- Numbers
- Phone numbers
- Social Security numbers
- Strings
- ZIP codes

The following example modifies your data model to insert two data validators: one for the ZIP code field and one for the phone number field:

```
<!-- Define data model. -->
<mx:Model id="myFormModel">
  <zipCodeModel>{zipCode.text}</zipCodeModel>
  <phoneNumberModel>{phoneNumber.text}</phoneNumberModel>
</mx:Model>

<!-- Define validators. -->
<mx:ZipCodeValidator field="myFormModel.zipCodeModel" />
<mx:PhoneNumberValidator field="myFormModel.phoneNumberModel" />
```

When the user enters data into the `zipCode` form field, Flex automatically copies that data to the data model. As part of that operation, Flex uses the associated data validator to verify that the input data is a valid ZIP code. If the ZIP code is valid, Flex only performs the copy. If the input data is invalid, Flex draws a red box around the associated form control. In addition, if the user mouses over the control, Flex displays an error message using a `ToolTip`, as the following figure shows:



The error `ToolTip` disappears as soon as the user moves the mouse away from the form control, or after a short delay. For more information on using validators, see [Chapter 28, “Managing Data in Flex,”](#) on page 629.

Populating a Form control from a data model

Another use for data models is to include data in the model to populate values of form fields. The following example shows a form that reads static data from a data model to obtain the value for a form field:

```
<!-- Define data model. -->
<mx:Model id="myFormModel">
  <fName>{firstName.text}</fName>
  <lName>{lastName.text}</lName>
  <department>Accounting</department>
  ...
</mx:Model>

<mx:Form>
```

```

<mx:FormItem label="Department" >
    <mx:TextInput id="dept" text="{myFormModel.department}" />
</mx:FormItem>

...
</mx:Form>

```

This data is considered static because the form always shows the same value for the department. You could also create a dynamic data model that takes the value of the department field from a web service, or calculates it based on user input.

For more information on data models, see [Chapter 28, “Managing Data in Flex,” on page 629](#).

Submitting data to a server

Form data is typically processed on a server, not locally on the client. Therefore, the submit handler must have a mechanism for packing the form data for transfer to the server, then handle any results returned back from the server. In Flex, you typically use a web service, HTTP service, or remote Java object to pass data to the server.

One additional aspect of submitting form data is building logic into your submit function to control navigation of your application when the submit succeeds, or when it fails. Upon a successful submit, you typically navigate to an area of your application that displays the results. If the submit fails, you can return control to the form so that the user can fix any errors.

The following example adds a web service to process form input data. In this example, the user enters a ZIP code, then selects the Submit button. After performing any data validation, the submit handler calls the web service to obtain the city name, current temperature, and forecast for the ZIP code.

```

<!-- Define the web service connection. Note that the specified WSDL URI
is not functional. -->
<mx:WebService id="WeatherService" wsdl="/ws/WeatherService?wsdl">
    <mx:operation name="GetWeather" >
        <mx:request>
            <ZipCode>{zipCode.text}</ZipCode>
        </mx:request>
    </mx:operation>
</mx:WebService>

<mx:Script>
    <![CDATA[
        function processValues()
        {
            // Check to see if ZIP code is valid.
            WeatherService.GetWeather.send();
        }
    ]]>
</mx:Script>

<mx:Form>
    <mx:FormItem label="Zip Code" >
        <mx:TextInput id="zipCode" width="200" text="Zip code please?" />
        <mx:Button width="60" label="Submit" click="processValues()" />
    </mx:FormItem>
</mx:Form>

```

```

        </mx:FormItem>
    </mx:Form>

    <mx:VBox>
        <mx:TextArea text="{WeatherService.GetWeather.result.CityShortName}"/>
        <mx:TextArea text="{WeatherService.GetWeather.result.CurrentTemp}"/>
        <mx:TextArea
            text="{WeatherService.GetWeather.result.DayForecast[0].Forecast}"/>
    </mx:VBox>

```

This example binds the form's input `zipCode` field directly to the `ZipCode` field of the web service request. To display the results from the web service, you bind its results to controls in a `VBox` container.

You have a great deal of flexibility when passing data to a web service. For example, you might modify this example to bind the input form field to a data model, and then bind the data model to the web service request. For more information on using web services, see [Chapter 30, "Using Data Services,"](#) on page 655.

You can also add event handlers for the web service to handle both a successful call to the web service, using the `load` event, and a call that generates an error, using the `fault` event. An error condition might cause you to display a message to the user with a description of the error. A successful result might have you navigate to another section of your application.

The following example adds a `load` event and a `fault` event to the form. In this example, the form is defined as one child of a `ViewStack` container, and the form results are defined as a second child of the `ViewStack` container:

```

<mx:WebService id="WeatherService"
    wsdl=
        "http://weather.unisysfsp.com/PDCWebService/WeatherServices.asmx?WSDL"
    result="successfulCall()"
    fault="errorCall();" >
    <mx:operation name="GetWeather" >
        <mx:request>
            <ZipCode>{zipCode.text}</ZipCode>
        </mx:request>
    </mx:operation>
</mx:WebService>

<mx:Script>
    <![CDATA[
        function processValues()
        {
            // Check to see if ZIP code is valid.
            WeatherService.GetWeather.send();
        }
        function successfulCall()
        {
            vs1.selectedIndex=1;
        }
        function errorCall()
        {

```

```

        mx.controls.Alert.show("Web service failed!", "Alert Box",
                               mx.controls.Alert.OK);
    }
    ]]>
</mx:Script>

<mx:ViewStack id="vs1" >
    <mx:Form>
        <mx:FormItem label="Zip Code" >
            <mx:TextInput id="zipCode" width="200" text="Zip code please?" />
            <mx:Button width="60" label="Submit" click="processValues()" />
        </mx:FormItem>
    </mx:Form>

    <mx:VBox>
        <mx:TextArea text="{WeatherService.GetWeather.result.CityShortName}" />
        <mx:TextArea text="{WeatherService.GetWeather.result.CurrentTemp}" />
        <mx:TextArea
            text="{WeatherService.GetWeather.result.DayForecast[0].Forecast}" />
    </mx:VBox>
</mx:ViewStack>

```

Upon a successful call to the web service, the `successfulCall()` function switches the current `ViewStack` child to the `VBox` container to show the returned results. An error from the web service displays an `Alert` box, but does not change the current child of the `ViewStack` container, so the form remains visible, letting the user fix any input errors.

You have many options for handling navigation in your application based on the results of the submit. The previous example used a `ViewStack` container to handle navigation. You might also choose to use a `TabNavigator` container or `Accordion` container for this same purpose.

In some applications, you might choose to embed the form in a `TitleWindow` container. A `TitleWindow` container is a pop-up window that appears above Flash Player drawing surface. In this scenario, users enter form data and submit the form from the `TitleWindow` container. A successful submit closes the `TitleWindow` container and displays the results in another area of your application; a failure displays an error message and leaves the `TitleWindow` container visible.

Another type of application might use a dashboard layout, where you have multiple panels open on the dashboard. Submitting the form could cause another area of the dashboard to update with results, while a failure could display an error message.

For more information on the `TabNavigator`, `Accordion`, and `TitleWindow` containers, see [Chapter 7, “Using Navigator Containers,” on page 247](#).

Form container syntax

You use the `<mx:Form>` tag to define a Form container. The Form container inherits all of the properties of the classes `MovieClip`, `UIObject`, `UIComponent`, `View`, and `Container`. For a list of these properties, see [“Configuring containers” on page 185](#). For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*.

This container also defines the following optional properties:

You use the `<mx:FormHeading>` tag to define a heading. The `<mx:FormHeading>` tag takes all the properties of the `<mx:Canvas>` tag, and the optional properties described in the following table:

| Property | Type | Use | Descriptions |
|-------------------------|--------|----------|--|
| <code>label</code> | String | Property | Specifies the label text. |
| <code>labelWidth</code> | String | Property | Specifies the label width for the heading. |

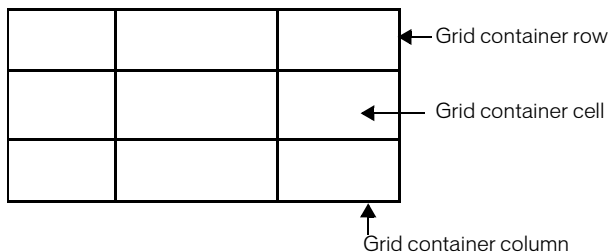
You use the `<mx:FormItem>` tag to define an item in a Form container. For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*. The `<mx:FormItem>` tag takes all the properties of the `<mx:Canvas>` tag, except `marginLeft`, and the optional properties defined in the following table:

| Property | Type | Use | Description |
|--------------------------|---------|----------|---|
| <code>direction</code> | String | Property | Specifies the direction of the children. Possible values are <code>vertical</code> , or <code>horizontal</code> . The default value is <code>vertical</code> . |
| <code>label</code> | String | Property | Specifies the label of the <code>FormItem</code> container. |
| <code>labelObject</code> | Object | Property | A read-only property that lets you access the label of the <code>FormItem</code> container. |
| <code>required</code> | Boolean | Property | Specifies that the <code>FormItem</code> children require user input, if <code>true</code> . The default value is <code>false</code> , indicating that input is not required. |

For more information on the syntax of the `<mx:Canvas>` tag, see [“Canvas container syntax” on page 209](#).

Grid layout container

You use a Grid layout container to arrange children as rows and columns of cells, much like an HTML table. The following figure shows a Grid container that consists of nine cells arranged in a 3x3 pattern:



You can put zero or one child in each cell of a Grid container. If you want to put multiple children in a cell, put a container in the cell, then add children to the container. The height of all cells in a row is the same, but each row can have a different height. The width of all cells in a column is the same, but each column can have a different width.

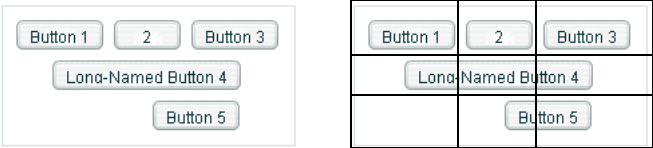
You can define a different number of cells for each row or each column of the Grid container. In addition, a cell can span multiple columns and/or multiple rows of the container.

A Grid container has the following default properties:

| Property | Default |
|--------------------------------|--|
| Preferred height of each row | The preferred height of the tallest cell in the row. |
| Preferred width of each column | The preferred width of the widest cell in the column. |
| Row sizing rules | Rows do not resize by default. You can make them resize according to their parent container's size by setting <code>width</code> and <code>height</code> to percentage values. |
| Column sizing rules | Columns do not resize by default. You can make them resize according to their parent container's size by setting <code>width</code> and <code>height</code> to percentage values. |
| Container resizing rules | Grid layout containers do not resize by default. You can make them resize according to their parent container's size by setting <code>width</code> and <code>height</code> to percentage values. |
| Default margins | Top, bottom, left, right = 0 pixels. |

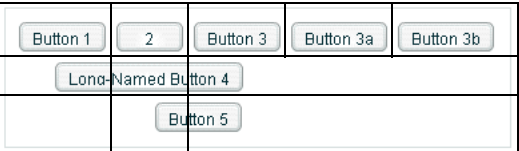
Grid layout container example

The following figure shows an example Grid container with three rows and three columns:



The figure on the left shows how the Grid container appears in Flash Player. The figure on the right shows the Grid container with borders overlaying it to illustrate the configuration of the rows and columns. In this example, the buttons in the top row each occupy a single cell. The button in the second row spans three columns, and the button in the third row spans the second and third columns.

You do not have to define the same number of cells for every row of a Grid container. The following figure shows a Grid container with five cells defined for the first row, and three cells for the next two rows:



You use the following MXML tags to create a Grid container:

| Tag | Description |
|----------------------------------|---|
| <code><mx:Grid></code> | Defines a Grid container. A Grid container has one or more rows. |
| <code><mx:GridRow></code> | Defines a grid row. A row has one or more cells. |
| <code><mx:GridItem></code> | Defines a grid cell. The <code><mx:GridItem></code> tag can contain any number of child tags. |

The following MXML code creates a Grid container with three rows and three columns:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

  <mx:Grid id="myGrid">

    <!-- Define Row 1. -->
    <mx:GridRow id="row1" >
      <!-- Define the first cell of Row 1. -->
      <mx:GridItem>
        <mx:Button label="Button 1"/>
      </mx:GridItem>
      <!-- Define the second cell of Row 1. -->
      <mx:GridItem>
        <mx:Button label="2"/>
      </mx:GridItem>
      <!-- Define the third cell of Row 1. -->
      <mx:GridItem>
        <mx:Button label="Button 3"/>
      </mx:GridItem>
    </mx:GridRow>

    <!-- Define Row 2. -->
    <mx:GridRow id="row2" >
      <!-- Define a single cell to span three columns of Row 2. -->
      <mx:GridItem colSpan="3" horizontalAlign="center" >
        <mx:Button label="Long-Named Button 4"/>
      </mx:GridItem>
    </mx:GridRow>

    <!-- Define Row 3. -->
    <mx:GridRow id="row3">
      <!-- Define an empty first cell of Row 3. -->
      <mx:GridItem/>
      <!-- Define a cell to span columns 2 and 3 of Row 3. -->
      <mx:GridItem colSpan="2" horizontalAlign="center">
        <mx:Button label="Button 5" />
      </mx:GridItem>
    </mx:GridRow>

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

To modify this example to include five buttons across the top row, you modify the first

`<mx:GridRow>` tag as follows:

```

<!-- Define Row 1. -->
<mx:GridRow id="row1">
  <!-- Define the first cell of Row 1. -->
  <mx:GridItem>
    <mx:Button label="Button 1"/>
  </mx:GridItem>
  <mx:GridItem>
    <mx:Button label="2"/>
  </mx:GridItem>
  <mx:GridItem>
    <mx:Button label="Button 3"/>
  </mx:GridItem>
  <mx:GridItem>
    <mx:Button label="Button 3a"/>
  </mx:GridItem>
  <mx:GridItem>
    <mx:Button label="Button 3b"/>
  </mx:GridItem>
</mx:GridRow>

```

Setting the row and column span

The following figure shows a modification to the previous example, where Button 3a now spans two rows, Button 3b spans three rows, and Button 5 spans three columns:



This example also adds flex properties to Buttons 3a, 3b, and 5 to make them resizable, as the following example code shows:

```

<mx:Grid>
  <!-- Define Row 1. -->
  <mx:GridRow id="row1" height="33%">
    <!-- Define the first cell of Row 1. -->
    <mx:GridItem>
      <mx:Button label="Button 1"/>
    </mx:GridItem>
    <mx:GridItem>
      <mx:Button label="2"/>
    </mx:GridItem>
    <mx:GridItem>
      <mx:Button label="Button 3"/>
    </mx:GridItem>
    <mx:GridItem rowSpan="2" >
      <mx:Button label="Button 3a" height="100%" />
    </mx:GridItem>
    <mx:GridItem rowSpan="3" >
      <mx:Button label="Button 3b" height="100%" />
    </mx:GridItem>
  </mx:GridRow>

```



```

<!-- Define Row 2. -->
<mx:GridRow id="row2" height="33%">
    <!-- Define a single cell to span three columns of Row 2 -->
    <mx:GridItem colSpan="3" horizontalAlign="center">
        <mx:Button label="Long-Named Button 4"/>
    </mx:GridItem>
</mx:GridRow>

<!-- Define Row 3. -->
<mx:GridRow id="row3" height="33%">
    <!-- Define an empty first cell of Row 3. -->
    <mx:GridItem/>
    <!-- Define a cell to span columns 2 and 3 and 4 of Row 3. -->
    <mx:GridItem colSpan="3" >
        <mx:Button label="Button 5 expands across 3 columns" width="75%" />
    </mx:GridItem>
</mx:GridRow>

</mx:Grid>

```

If you had omitted the height from Buttons 3a and 3b, Flex would have set the buttons to their preferred height, so they would not appear to span the rows. By adding the percentage width property to Button 5, you cause it to expand to the full width of the three columns, not to its preferred width, which is the width of its text.

Even though the second row contains only a single `<mx:GridItem>` tag that defines a cell spanning three columns, Flex automatically adds cells to the grid to allow Buttons 3a and 3b to expand down into the row. The same is true for row three, which only defines four cells.

Sizing and positioning a child within a Grid container cell

The preferred size of each grid cell is determined by the preferred height of the tallest cell in the row and the preferred width of the widest cell in the column. If the preferred size of the child is larger than the cell, the child is clipped at the cell boundaries.

If the child's preferred width or preferred height is smaller than the cell, the default horizontal alignment of the child in the cell is `left` and the default vertical alignment is `top`. You can use the `horizontalAlign` and `verticalAlign` properties of the `<mx:GridItem>` tag to control positioning of the child.

If the child is resizable, as determined by its `height` and `width` properties, the child is enlarged or shrunk to fit the size of the cell.

Setting the spacing between rows and columns

The `horizontalGap` and `verticalGap` properties of the `<mx:Grid>` tag determine the spacing between rows and columns of the Grid layout container. By default, Flex sets both gaps to 6 pixels.

Note: Flex ignores both the `horizontalGap` and `verticalGap` properties for the `<mx:GridRow>` tag and the `<mx:GridItem>` tag.

Grid container syntax

You use the `<mx:Grid>` tag to define a Grid container. The `<mx:Grid>` tag takes all the properties of the `<mx:Box>` tag, except the `direction` property. For more information on the syntax of the `<mx:Box>` tag, see [“Box container syntax” on page 210](#).

You use the `<mx:GridRow>` tag to define each row, and it must be a child of the `<mx:Grid>` tag. The `<mx:GridRow>` tag can hold any number of `<mx:GridItem>` child tags. The `<mx:GridRow>` tag takes all the properties of the `<mx:Grid>` tag, but ignores both the `horizontalGap` and `verticalGap` properties.

You use the `<mx:GridItem>` tag to define a row cell, and it must be a child of the `<mx:GridRow>` tag. The `<mx:GridItem>` tag takes all the properties of the `<mx:Grid>` tag, but ignores the `horizontalGap` and `verticalGap` properties. In addition, the `<mx:GridItem>` tag takes the optional properties described in the following table:

| Property | Type | Use | Descriptions |
|----------------------|--------|----------|---|
| <code>rowSpan</code> | Number | Property | Specifies the number of rows of the Grid container spanned by the cell. The default value is 1. You cannot extend a cell past the number of rows in the Grid container. |
| <code>colSpan</code> | Number | Property | Specifies the number of columns of the Grid container spanned by the cell. The default value is 1. You cannot extend a cell past the number of columns in the Grid container. |

Panel layout container

A Panel layout container includes a title bar, a title, a status message, a border, and a content area for its children. Typically, you use Panel containers to wrap self-contained application modules. For example, you could define several Panel containers in your application where one Panel container holds a form, a second holds a shopping cart, and a third holds a catalog.

The following figure shows a Panel container with a Form container as its child:

The figure shows a window titled "My Application" containing a "Billing Information" form. The form has the following elements:

- First Name:
- Last Name:
- Address:
- City / State:
- ZIP Code:
- Country:
- Submit:

A Panel container has the following default properties:

| Property | Default |
|------------------------|---|
| Preferred size | Height is large enough to hold all of its children at the preferred height of the children, plus any vertical gap between the children, plus the top and bottom container margins. Width is the larger of the preferred width of the widest child plus the left and right container margins, or the width of the title text. |
| Container layout rules | The Panel container is a subclass of the VBox container so it lays out its children in a single vertical column. |
| Margins | Top, bottom, left, right = 4 pixels. |

Creating a Panel container

You define a Panel container in MXML using the `<mx:Panel>` tag, as the following example shows. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

The following example defines a Panel container that contains a form as the top-level container in your application. In this example, the Panel container provides you with a mechanism for including a title bar, as in a standard GUI window.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

    <mx:Panel id="myPanel" title="My Application" />

        <mx:Form width="300" >
            <mx:FormHeading label="Billing Information" />
            ...
        </mx:Form>

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

Adding a ControlBar container to a Panel container

You can use the ControlBar container with a Panel container to hold components that can be shared by the other children in the Panel container. For a product catalog, the ControlBar container can hold the Flex controls to specify quantity and to add an item to a shopping cart, as the following figure shows:



You specify the `<mx:ControlBar>` tag as the last child tag of an `<mx:Panel>` tag, as the following example shows:

```
<mx:Panel title="My Application" marginTop="10" marginBottom="10"
  marginLeft="10" marginRight="10" width="300">

  <mx:HBox width="100%">
    <!-- Area for your catalog. -->
  </mx:HBox>

  <mx:ControlBar width="100%">
    <mx:Label text="Quantity" />
    <mx:NumericStepper id="myNS"/>
    <!-- Use Spacer to push Button control to the right. -->
    <mx:Spacer width="100%" />
    <mx:Button label="Add to Cart" click="addToCart()" />
  </mx:ControlBar>
</mx:Panel>
```

For more information on the ControlBar container, see [“ControlBar layout container” on page 211](#).

Panel container syntax

You use the `<mx:Panel>` tag to define a Panel container. For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*. The Panel container inherits all of the properties of the MovieClip, UIObject, UIComponent, View, Container, Box, and VBox classes. For more information, see [“Box container syntax” on page 210](#).

This container also defines the following optional properties:

| Property | Type | Use | Description |
|------------------------|--------|----------|---|
| status | String | Property | Specifies text in the status area of the title bar. |
| title | String | Property | Specifies the title text. |
| statusStyleDeclaration | String | Property | Specifies a style sheet definition to configure the status area of the control. |
| titleStyleDeclaration | String | Property | Specifies the style sheet definition for the title text. |

Note: Note: The `borderStyle` style is not supported on the Panel container. To make solid border Panels, set the `borderThickness` property, and set the `dropShadow` property to `false` if desired.

Tile layout container

A Tile layout container lays out its children in one or more vertical columns or horizontal rows, starting new rows or columns as necessary. The `direction` property determines the layout. The possible values for the `direction` property are `vertical` for a column layout and `horizontal` (default) for a row layout.

All Tile container cells have the same size. Flex arranges the cells of a Tile container in a square grid, where each cell holds a single child component. For example, if you define 16 children in a Tile layout container, Flex lays it out four cells wide and four cells high. If you define 13 children, Flex still lays it out four cells wide and four cells high, but leaves the last three cells in the fourth row empty.

The following figure shows examples of horizontal and vertical Tile containers:



A Tile container has the following default properties:

| Property | Default |
|-----------------------------|--|
| Direction | Horizontal |
| Preferred size of all cells | Height is the preferred height of the tallest child; width is the preferred width of the widest child. All cells have the same preferred size. |

| Property | Default |
|----------------------------------|---|
| Preferred size of Tile container | Flex computes the square root of the number of children, and rounds up to the nearest integer. For example, if there are 26 children, the square root is 5.1, which is rounded up to 6. Flex then lays out the Tile container in a 6x6 grid. The preferred height of the Tile container is equal to (tile cell preferred height) * (rounded square root of the number of children). The preferred width is equal to (tile cell preferred width) * (rounded square root of the number of children). |
| Minimum size of Tile container | The preferred size of a single cell. Flex always allocates enough space to display at least one cell. |
| Default margins | Top, bottom, left, right = 0 pixels. |

Tile layout container example

You define a Tile container in MXML using the `<mx:Tile>` tag. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an `ActionScript` block. The following example creates the horizontal Tile container shown in the previous figure:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

    <mx:Tile id="myFlow" direction="horizontal" borderStyle="solid"
        marginTop="10" marginBottom="10" marginRight="10" marginLeft="10"
        verticalGap="15" horizontalGap="10" >
        <mx:TextInput id="text1" text="1" height="50" width="75" />
        <mx:TextInput id="text2" text="2" height="50" width="100"/>
        <mx:TextInput id="text3" text="3" height="50" width="75"/>
        <mx:TextInput id="text4" text="4" height="50" width="75"/>
        <mx:TextInput id="text5" text="5" height="50" width="75" />
    </mx:Tile>

</mx:Application>
```

Sizing and positioning a child in a Tile container

Flex sets the preferred size of each Tile cell to the height of the tallest child and the width of the widest child. All cells have the same preferred size. If the preferred size of a child is larger than the cell, for example, if you used `tileHeight` and `tileWidth` to explicitly size the cells, the child is automatically sized to fit inside the cell boundaries. This may lead content inside the control to be clipped; for instance, the label of a button might be clipped even though the button itself fits into the cell.

If the child's preferred width or preferred height is smaller than the cell, the default horizontal alignment of the child in the cell is `left` and the default vertical alignment is `top`. You can use the `horizontalAlign` and `verticalAlign` properties of the `<mx:Tile>` tag to control the positioning of the child.

If the child is resizable, the child is enlarged or shrunk to fit the size of the cell. In the example in [“Tile layout container example” on page 238](#), the `TextInput` control named `text2` has a width of 100 pixels; therefore, the preferred width of all `Tile` cells is 100 pixels. If you want all other `TextInput` controls to increase in size to the preferred width of the cells, set the `width` property to a percentage value, as the following example shows:

```
<mx:Tile id="myFlow" direction="horizontal" borderStyle="solid"
    marginTop="10" marginBottom="10" marginRight="10" marginLeft="10"
    verticalGap="15" horizontalGap="10" width="100%" tileWidth="100">
    <mx:TextInput id="fname" text="1" height="50" width="100%" />
    <mx:TextInput id="lname" text="2" height="50" width="100%" />
    <mx:TextInput id="addr1" text="3" height="50" width="100%" />
    <mx:TextInput id="addr2" text="4" height="50" width="100%" />
    <mx:TextInput id="addr3" text="5" height="50" width="100%" />
</mx:Tile>
```

Note: Set the width and height size to a percentage of the tile cell, rather than to a percentage of the `Tile` container. Even though it isn't an explicitly defined container, the cell acts as the parent of the components within the `Tile` container.

Tile container syntax

You use the `<mx:Tile>` tag to create a `Tile` container. The `<mx:Tile>` tag inherits all of the properties of the classes `MovieClip`, `UIObject`, `UIComponent`, `View`, and `Container`. For a list of these properties, see [“Configuring containers” on page 185](#). For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*.

This container also defines the following optional properties:

| Property | Type | Use | Description |
|-------------------------|--------|----------|--|
| <code>direction</code> | String | Property | Specifies the orientation of the container. Possible values are <code>horizontal</code> and <code>vertical</code> . The default value is <code>horizontal</code> . |
| <code>tileHeight</code> | Double | Property | Specifies the explicit height of all cells, overriding the default of the preferred height of the tallest child. |
| <code>tileWidth</code> | Double | Property | Specifies the explicit width of all cells, overriding the default of the preferred width of the widest child. |

TitleWindow layout container

A `TitleWindow` layout container defines a pop-up window that consists of a title bar, a caption, a border, and a content area for its children. For example, you can include a form in a `TitleWindow` container. When the user completes the form, you can close the `TitleWindow` container programmatically, or let the user close it by using the Close button in the upper-right corner of the window.

The `TitleWindow` container is almost the same as the `Panel` container. The only difference is that the `TitleWindow` is designed to work as a pop-up window. The `Panel` container is designed to work as a static window.

The following figure shows a TitleWindow container with a Form container as its child:

The image shows a window titled "My Application" with a close button (X) in the top right corner. Inside the window is a form titled "Billing Information". The form contains the following fields and controls:

- First Name: A text input field.
- Last Name: A text input field.
- Address: A text input field.
- City / State: A text input field followed by a small dropdown arrow.
- ZIP Code: A text input field.
- Country: A small dropdown arrow.
- Submit: A button labeled "Submit".

A TitleWindow container has the following default properties:

| Property | Default |
|----------------|---|
| Preferred size | Height is large enough to hold all of its children at the preferred height of the children, plus any vertical gap between the children, plus the top and bottom container margins. Width is the larger of the preferred width of the widest child plus the left and right container margins, or the width of the title text. |
| Margins | Top, bottom, left, right = 4 pixels. |

Creating a pop-up TitleWindow container

A pop-up TitleWindow container appears on top of your Flex application in Flash Player. A pop-up TitleWindow container can be modal, which means that it takes all keyboard and mouse input until it is closed, or nonmodal, which means other windows can accept input while the pop-up window is still open. Users can move a pop-up TitleWindow container by using the mouse to drag it around the screen.

Although the contents and behavior of the window are defined in the TitleWindow container, to create a pop-up window, you may have to use methods of the PopUp Manager. You can create modal pop-up windows directly within a TitleWindow container, or you can use the PopUp Manager. You can create nonmodal windows only by using the PopUp Manager. For more information, see [“Creating a pop-up TitleWindow container using the PopUp Manager” on page 243](#).

The Application container defines the following method that you can use to open a TitleWindow container directly:

```
popupWindow(class:Object, initobj:Object) : MovieClip;
```

where:

class Specifies a reference to the class of object you want to create.

initobj Specifies an optional object containing initialization properties. This argument is optional.

This method returns a MovieClip object that you can cast to a TitleWindow object.

Creating a TitleWindow using the popupWindow() method makes the deletePopUp() method available to the TitleWindow container. You use this method to close the pop-up window.

One of the most common ways of creating a TitleWindow container is to define it as a custom MXML component. You define the TitleWindow container, and all of its children, in the custom component, then use the popupWindow() method to open the TitleWindow container.

The following example code shows the definition of the custom component:

```
<?xml version="1.0"?>

<mx:TitleWindow xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:Label text="Enter User name" />
  <mx:TextInput width="100"/>
  <mx:Label text="Enter Password" />
  <mx:TextInput width="100" password="true"/>
  <mx:HBox>
    <mx:Button click="submitForm();" label="OK" />
    <mx:Button click="deletePopUp()" label="Cancel" />
  </mx:HBox>
</mx:TitleWindow>
```

This file, named MyLogin.mxml, defines a TitleWindow container using the <mx:TitleWindow> tag. The child of the TitleWindow container is a VBox container that defines two TextInput controls, for user name and password, and two Button controls, for submitting the form and for closing the TitleWindow container. This example does not include the code for the submitForm() event handler.

You create the TitleWindow container from the Main.mxml file, as the following example code shows:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">

  <mx:Script>
    <![CDATA[
      // Import TitleWindow and Application.
      import mx.containers.TitleWindow;
      import mx.core.Application;

      function showLogin()
      {
        // Create the TitleWindow container and cast it to a TitleWindow.
```

```

        var pop1:TitleWindow = TitleWindow(popupWindow(MyLogin));
    }
]]>
</mx:Script>

<mx:VBox width="300" height="500">
    <mx:Button click="showLogin()" label="Login" />
</mx:VBox>

</mx:Application>

```

In this example, when the user selects the Login button, the event handler for the `click` event uses the `popupWindow()` method to create a `TitleWindow` container, passing to it the name of the `MyLogin.mxml` file as the class name.

By default, the `TitleWindow` container includes a close button in the upper right corner, similar to dialog boxes in a GUI environment. Users can close the `TitleWindow` container by selecting the close button. In addition, the `TitleWindow` broadcasts a `click` event when the user selects the close button. You can optionally specify an event handler for the `click` event, as the following code shows:

```

<mx:TitleWindow click="handleClick(event)"
    xmlns:mx="http://www.macromedia.com/2003/mxml" >

```

To hide the close button, you set the `closeButton` property in the `<mx:TitleWindow>` tag to `false`, as the following code shows:

```

<mx:TitleWindow closeButton="false"
    xmlns:mx="http://www.macromedia.com/2003/mxml" >

```

Passing optional arguments to the `popupWindow()` method

The `popupWindow()` method takes an optional *initobj* argument that lets you pass initialization properties to the custom component. The custom component is a `TitleWindow` container, so the same properties that you can set in MXML for the `TitleWindow` container are available through this argument.

For example, the following call to the `popupWindow()` method passes `height`, `width`, and `title` properties:

```

var pop1:TitleWindow = TitleWindow(popupWindow(MyLogin,
    {title:'Confirmation', width:300, height:200}));

```

You can include the `closeButton` property in the *initobj* argument, but you cannot specify the handler for the `click` event. However, you can set it programmatically, as the following example shows:

```

//Create the TitleWindow container.
var pop1:TitleWindow = TitleWindow(popupWindow(MyLogin,
    {title:'Confirmation', width:300, height:200, closeButton:true }));

//Create the event handler.
var eventHandlerObj=new Object();
eventHandlerObj.click=function(evt)
{

```

```

    evt.target.deletePopUp();
}
//Register the event handler with the TitleWindow container.
pop1.addEventListener("click",eventHandlerObj);

```

Creating a pop-up TitleWindow container using the PopUp Manager

You can use the PopUp Manager to create either a modal or a nonmodal pop-up TitleWindow container. The PopUp Manager provides the following method, which you use in the TitleWindow container:

```
createPopUp(parent:MovieClip, class:Object [, modal:Boolean, initobj:Object,
    outsideEvents:Boolean]) : MovieClip
```

where:

parent Specifies a reference to a window to popup over.

class Specifies a reference to the class of object you want to create.

modal Specifies an optional Boolean value indicating whether the window is modal (true) or not (false). The default value is false. This argument is optional.

initobj Specifies an optional object containing initialization properties. This argument is optional.

outsideEvents Specifies an optional Boolean value indicating whether an event is triggered if the user clicks outside the window (true) or not (false). The default value is false. This argument is optional.

Creating a TitleWindow using the createPopUp() method makes the deletePopUp() method available to the TitleWindow container. You use this method to close the pop-up.

In this example, you define the TitleWindow container, and all of its children, in the custom component, and then use the createPopUp() method to open the TitleWindow container. For a definition of the custom component, see [“Creating a pop-up TitleWindow container” on page 240](#).

You create the TitleWindow container from the Main.mxml file, your main application file, and reference the MyLogin component (MyLogin.mxml) within the createPopUp() call:

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">

    <mx:Script>
        <![CDATA[
            import mx.containers.TitleWindow;
            import mx.managers.PopUpManager;

            function showLogin()
            {
                // Create the pop-up TitleWindow container.
                var pop1:TitleWindow =
                    TitleWindow(PopUpManager.createPopUp(this, MyLogin));
            }
        ]]>

```

```

</mx:Script>

<mx:VBox id="myVBox" width="300" height="500">
  <mx:Button click="showLogin()" label="Login" />
</mx:VBox>

</mx:Application>

```

In this example, when the user selects the Login button, the event handler for the `click` event uses the `createPopUp()` method to create a `TitleWindow` container, passing to it the name of the `MyLogin.mxml` file as the class name.

If you want to allow users to close the `TitleWindow` using the close button when you create it using the `PopUp Manager`, you must define an event handler for the container's `click` event that calls the `deletePopUp()` method.

Call the `centerPopUp()` method of the `PopUp Manager` to center a `TitleWindow` container within another container. For example, to center the `TitleWindow` container in the previous example, modify `showLogin()`, as the following example shows:

```

function showLogin()
{
    // Create the pop-up TitleWindow container.
    var pop1:TitleWindow =
        TitleWindow(PopUpManager.createPopUp(this, MyLogin));
    pop1.centerPopUp(myVBox);
}

```

Passing optional arguments to the `createPopUp()` method

The `createPopUp()` method takes several optional arguments. To create a modal `TitleWindow` container, set the *modal* argument to `true`. When a `TitleWindow` is modal, you cannot select any other component while the window is open. The default value of *modal* is `false`.

You can use *initobj* argument of the `createPopUp()` method to pass initialization properties to the custom component. The custom component is a `TitleWindow` container, which means you can pass to it the same properties that you can set in MXML for the `TitleWindow` container. For an example, see [“Passing optional arguments to the `popupWindow\(\)` method” on page 242](#).

If you set *outsideEvents* to `true`, whenever a user clicks the mouse outside of the window, Flex broadcasts a `mouseDownOutside` event. The default value is `false`. To handle this event, you can create an event handler and register it as a listener, as the following example shows:

```

listenerObj = new Object();
listenerObj.mouseDownOutside = function()
{
    deletePopUp();
}
myTW.addEventListener("mouseDownOutside", listenerObj);

```

You can set *outsideEvents* to `true` even if you do not want to use the *initobj* argument, by passing `undefined` as its value, as the following example shows:

```

var pop1:TitleWindow =
    TitleWindow(PopUpManager.createPopUp(this, MyLogin, true, undefined, true));

```

Passing data to a pop-up TitleWindow container

To make the custom component that defines your TitleWindow container more flexible, you might want to pass data to it or return data from it. For example, the following application opens a pop-up TitleWindow container and passes to it a reference to a component in the Application container so that the custom component can write its results back to the container:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">

  <mx:Script>
    <![CDATA[
      import mx.containers.TitleWindow;
      import mx.core.Application;

      function showName()
      {
        var pop1:TitleWindow = TitleWindow(popupWindow(MyLogin_passdata,
          {title:'Enter Name', closeButton:true, parentComponent: this.TA1}));
      }
    ]]>
  </mx:Script>

  <mx:VBox width="300" height="500">
    <mx:Button click="showName()" label="Enter name" />
    <mx:TextArea id="TA1" />
  </mx:VBox>
</mx:Application>
```

The custom component in the following example defines a variable to hold the reference to the parent component, then uses that reference to update the parent component:

```
<?xml version="1.0"?>
<mx:TitleWindow xmlns:mx="http://www.macromedia.com/2003/mxml"
  click="deletePopUp()">

  <mx:Script>
    <![CDATA[
      var parentComponent;
      function submitData()
      {
        parentComponent.text="You entered: " + inputText.text;
      }
    ]]>
  </mx:Script>

  <mx:VBox>
    <mx:Label text="Enter your name" />
    <mx:TextInput id="inputText" width="100"/>
    <mx:Button label="Send Data" click="submitData(); this.deletePopUp();" />
  </mx:VBox>
</mx:TitleWindow>
```

For more information on passing data to a custom component, see Chapter 2, “Using MXML” in *Getting Started with Flex*.

TitleWindow container syntax

You use the `<mx:TitleWindow>` tag to define a TitleWindow container. The TitleWindow container inherits all of the properties of the classes MovieClip, UIObject, UIComponent, View, Container, and Panel. For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*.

This container also defines the following optional properties:

| Property | Type | Use | Description |
|--------------------------|---------|----------|---|
| <code>closeButton</code> | Boolean | Property | Specifies whether to display a Close button in the TitleWindow container. The default value is <code>true</code> . Set it to <code>false</code> to hide the Close button. Selecting the Close button generates a <code>click</code> event, but does not close the TitleWindow container. You must write a handler for the <code>click</code> event and close the TitleWindow from within it. |
| <code>click</code> | | Event | Broadcast when the user selects the close button. |

CHAPTER 7

Using Navigator Containers

Navigator containers control user movement, or navigation, among multiple children where the children are other containers. The individual child containers of the navigator container oversee the layout and positioning of their children; the navigator container does not oversee layout and positioning.

This chapter describes navigator containers, and syntax, and contains samples using the navigator containers.

Contents

| | |
|---|-----|
| About navigator containers | 247 |
| ViewStack navigator container | 248 |
| LinkBar navigator container | 253 |
| TabBar navigator container | 255 |
| TabNavigator container | 259 |
| Accordion navigator container | 263 |

About navigator containers

A navigator container controls user movement through a group of child containers. For example, a TabNavigator container lets you select the visible child container using a set of tabs.

Note: The direct children of a navigator container must be containers, either layout or navigator containers. You cannot directly nest a control within a navigator; controls must be children of a child container of the navigator container.

Macromedia Flex provides the following navigator containers:

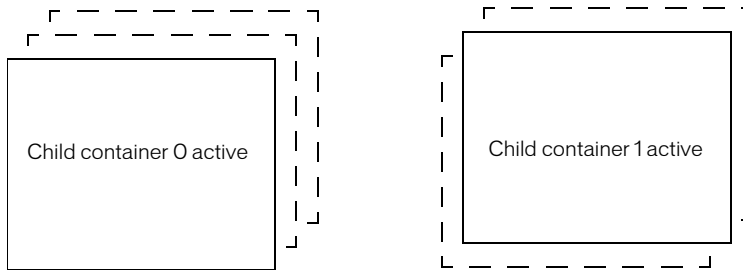
- ViewStack
- LinkBar
- TabNavigator
- TabBar
- Accordion

The following sections describes how to use each of the Flex navigator containers.

ViewStack navigator container

A ViewStack navigator container is made up of a collection of child containers *stacked* on top of each other with only one container visible, or *active*, at a time. The ViewStack container does not define a built-in mechanism for users to switch the currently active container; you must use a LinkBar or TabBar navigator container or build the logic yourself in ActionScript to let users change the currently active child. For example, you can define a set of Button controls that switch among the child containers.

The following figure shows a representation of a ViewStack container:



The figure on the left shows a ViewStack container with the first child active. The index of a child in a ViewStack container is zero-based, 0, 1, 2, ... , $n - 1$, where n is the number of child containers. The figure on the right shows the ViewStack container with the second child container active.

A ViewStack container has the following default properties:

| Property | Default |
|--------------------------|--|
| Preferred size | The width and height of the currently active child. |
| Container resizing rules | <p>ViewStack containers are only sized once to fit the size of the first child container by default. They do not resize when you navigate to other child containers by default. To force ViewStack containers to resize when you navigate to a different child container, set the <code>resizeToContent</code> property to <code>true</code>.</p> <p>You can ensure that all of your child containers fit in the ViewStack container with the expected layout by setting the <code>width</code> and <code>height</code> properties to explicit pixel values of sufficient size. You can also set both dimensions to 100% so the container fills all available space.</p> |
| Child sizing rules | Children are sized to their preferred size. The child is clipped if it is larger than the ViewStack container. If the child is smaller than the ViewStack container, it is aligned to the upper-left corner of the ViewStack container. |
| Default margins | Top, bottom, left, right = 0 pixels. |

Creating a ViewStack container

You use the following properties of the ViewStack container to control the active child container:

- `selectedIndex` The index of the currently active container if one or more child containers are defined; undefined if no child containers are defined. The index is zero-based, 0, 1, 2, ... , $n - 1$, where n is the total number of child containers in the ViewStack container. Set this property to the index of the container that you want active.

You can use the `selectedIndex` property of the `<mx:ViewStack>` tag to set the default active container when your application starts. The following example sets the index of the default active container to 1:

```
<mx:ViewStack id="myViewStack" selectedIndex=1 >
```

The following example uses ActionScript to set the `selectedIndex` property so that the active child container is the second container in the stack:

```
myViewStack.selectedIndex=1;
```

- `selectedChild` The identifier of the currently active container if one or more child containers are defined; undefined if no child containers are defined. Set this property in ActionScript to the identifier of the container that you want active.

You can only set this property in an ActionScript statement, not in MXML.

The following example uses ActionScript to set the `selectedChild` property so that the active child container is the child container with an identifier of `search`:

```
myViewStack.selectedChild=search;
```

- `numChildren` Contains the number of child containers in the ViewStack container. The ViewStack container inherits this property from the View class.

The following example uses the `numChildren` property in an ActionScript statement to set the active child container to the last container in the stack:

```
myViewStack.selectedIndex=myViewStack.numChildren-1;
```

The following example creates a ViewStack container with three child containers. The example also defines three Button controls that, when clicked, select the active child container:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

    <!-- Create a VBox container so the container for the buttons appears above
    the ViewStack container. -->
    <mx:VBox>

        <!-- Create an HBox container to hold the three buttons. -->
        <mx:HBox borderStyle="solid" >

            <!-- Define the three buttons. Each uses the child container identifier
            to set the active child container. -->
            <mx:Button id="searchButton" label="Search Screen"
                click="myViewStack.selectedChild=search;" />

            <mx:Button id="cInfoButton" label="Customer Info Screen"
                click="myViewStack.selectedChild=custInfo;" />

        </mx:HBox>
    </mx:VBox>

    <mx:ViewStack id="myViewStack">

        <!-- Define the three child containers. -->
        <mx:View id="searchScreen" data-bbox="100 100 300 200" />
        <mx:View id="custInfoScreen" data-bbox="350 100 550 200" />
        <mx:View id="otherScreen" data-bbox="600 100 800 200" />

    </mx:ViewStack>

</mx:Application>
```

```

        <mx:Button id="aInfoButton" label="Account Info Screen"
            click="myViewStack.selectedChild=accountInfo;" />

    </mx:HBox>

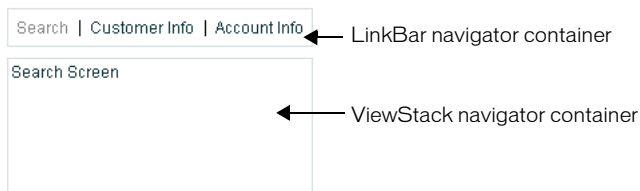
    <!-- Define the ViewStack and the three child containers and have it
         resize up to the size of the container for the buttons. -->
    <mx:ViewStack id="myViewStack" borderStyle="solid" width="100%" >
        <mx:Canvas id="search" label="Search">
            <mx:Label text="Search Screen" />
        </mx:Canvas>
        <mx:Canvas id="custInfo" label="Customer Info">
            <mx:Label text="Customer Info" />
        </mx:Canvas>
        <mx:Canvas id="accountInfo" label="Account Info">
            <mx:Label text="Account Info" />
        </mx:Canvas>
    </mx:ViewStack>

</mx:VBox>
</mx:Application>

```

When this example loads, the three Button controls appear, and the first child container defined in the ViewStack container is active. Select a Button control to change the active container.

You can also use a LinkBar or TabBar container to control the active child of a ViewStack container. A LinkBar or TabBar container determines the number of child containers in a ViewStack container, then creates a horizontal set of links or tabs that lets the user select the active child, as the following figure shows:



The items in the LinkBar container correspond to the values of the `label` property of each child of the ViewStack container, as the following example shows:

```

<mx:VBox>

    <!-- Create a LinkBar container to navigate the ViewStack container. -->
    <mx:LinkBar dataProvider="myViewStack" borderStyle="solid" />

    <!-- Define the ViewStack and the three child containers. -->
    <mx:ViewStack id="myViewStack" borderStyle="solid" width="100%">
        <mx:Canvas id="search" label="Search">
            <mx:Label text="Search Screen" />
        </mx:Canvas>
        <mx:Canvas id="custInfo" label="Customer Info">
            <mx:Label text="Customer Info" />
        </mx:Canvas>
    </mx:ViewStack>

```

```

<mx:Canvas id="accountInfo" label="Account Info">
    <mx:Label text="Account Info" />
</mx:Canvas>
</mx:ViewStack>

</mx:VBox>

```

You only have to provide a single property to the LinkBar or TabBar container, `dataProvider`, to specify the name of the ViewStack container associated with it. For more information on the LinkBar container, see [“LinkBar navigator container” on page 253](#). For more information on the TabBar container, see [“Accordion navigator container” on page 263](#).

Sizing the children of a ViewStack container

The default width and height of a ViewStack container is the width and height of the first child. A ViewStack container does not change size every time you change the active child.

You can use the following techniques to control the size of a ViewStack container so that it displays all of the components inside its children:

- Set explicit `width` and `height` properties for all children to the same fixed values.
- Set the `width` and `height` properties for all children to the same percentage values.
- Set `width` and `height` properties for the ViewStack container to 100%.

The technique that you use is based on your application and the content of your ViewStack container.

Order of initialization and creationComplete events

When you run your application, Flex broadcasts `initialize` and `creationComplete` events for the child containers of the ViewStack container, and for the children of the child containers. When Flex creates the ViewStack container, the order in which the events occur is as follows:

1. Broadcast the `initialize` event for the initially visible child container.
2. Broadcast the `initialize` event for all other child containers.
3. Broadcast the `initialize` event for the children of the initially visible child container.
4. Broadcast the `creationComplete` event for the initially visible child container.
5. Broadcast the `creationComplete` event for all other child containers.
6. Broadcast the `creationComplete` event for the children of the initially visible child container.

As you navigate the ViewStack container to make a different child container visible, Flex broadcasts `initialize` and `creationComplete` events for the children of the child container. The order in which the events occur for the children of a child container as it becomes visible is as follows:

1. Broadcast the `initialize` event for the children of the child container.
2. Broadcast the `creationComplete` event for the children of the child container.

You can also use the `childrenCreated` event with a `ViewStack` container. This event is broadcast after a container creates its children. This event lets you perform one-time initialization of a container's children, just after they get created.

For example, a `ViewStack` container creates all its immediate child containers, and all the children of its first visible child container. For the first visible child container, `childrenCreated` gets broadcast. Then, as the user moves to each additional child of the `ViewStack` container, the event gets dispatched for that container.

Applying behaviors to a `ViewStack` container

You can assign effects to the `ViewStack` container, or to its children. For example, if you want to assign the `WipeUp` effect to the `showEffect` property, Flex plays the effect once when the `ViewStack` first appears.

However, if you want to have the first child of the `ViewStack` container use a `WipeUp` effect for the `showEffect` property, and the second child use the `WipeDown` effect, assign the effects to the children of the `ViewStack` container, as the following example shows:

```
<mx:ViewStack id="myViewStack" borderStyle="solid" width="100%">
  <mx:Canvas id="search" label="Search" creationCompleteEffect="WipeUp"
    showEffect="WipeUp">
    <mx:Label text="Search Screen" />
  </mx:Canvas>
  <mx:Canvas id="custInfo" label="Customer Info" showEffect="WipeDown">
    <mx:Label text="Customer Info" />
  </mx:Canvas>
  <mx:Canvas id="accountInfo" label="Account Info" showEffect="WipeRight">
    <mx:Label text="Account Info" />
  </mx:Canvas>
</mx:ViewStack>
```

The `showEffect` property of a child of a `ViewStack` container is only triggered when the child's visibility changes from `false` to `true`. Therefore, the first child of the `ViewStack` container also includes a `creationCompleteEffect` property. This is necessary to trigger the effect when Flex first creates the component. If you omit `creationCompleteEffect` property, you do not see the `WipeUp` effect when the application starts.

ViewStack container syntax

You use the `<mx:ViewStack>` tag to define a `ViewStack` container. The `ViewStack` container inherits all of the properties of the classes `MovieClip`, `UIObject`, `UIComponent`, `View`, and `Container`. For a list of these properties, see [“Configuring containers” on page 185](#). For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*.

This container tag also defines the following properties:

| Property | Type | Use | Description |
|-------------------|---------|----------|--|
| historyManagement | Boolean | Property | Specifies to enable history management, <code>true</code> , or not, <code>false</code> . The default value is <code>false</code> . For more information, see Chapter 23, “Using the History Manager,” on page 569. |
| resizeToContent | Boolean | Property | Specifies whether the ViewStack container resizes when you navigate to a different child container. The default value is <code>false</code> . |
| selectedChild | String | Property | Specifies the name of the active container when the ViewStack container loads. The default is the identifier of the first child container defined in the ViewStack container. You can only set this property in an <code>ActionScript</code> statement, not in <code>MXML</code> . |
| selectedIndex | Number | Property | Specifies the index of the active container when the ViewStack container loads. Indexes are in the range of 0, 1, 2, ... , $n - 1$, where n is the number of child containers. The default value is 0, corresponding to the first child container defined in the ViewStack container. |
| change | | Event | • Broadcast when the current view changes. |

LinkBar navigator container

A LinkBar navigator container defines a horizontal row of Link controls that designate a series of link destinations. You typically use a LinkBar navigator container to control the active child container of a ViewStack container, or to create a stand-alone set of links.

The following figure shows an example of a LinkBar container that defines a set of links:



A LinkBar container has the following default properties:

| Property | Default |
|--------------------------|---|
| Preferred size | A width wide enough to contain all label text, plus any margins and separators, and the height of the tallest child. |
| Container resizing rules | LinkBar containers do not resize by default. Specify percentage sizes if you want your LinkBar to resize based on the size of its parent container. |
| Margins | Top, bottom, left, right = 2 pixels. |

Creating a LinkBar container

One of the most common uses of a LinkBar container is to control the active child of a ViewStack container. For an example, see [“Creating a ViewStack container” on page 249](#).

You can also use a LinkBar navigator container on its own to create a set of links in your application. In the following example, you define a `click` handler for the LinkBar container to respond to user input, and use the `dataProvider` property of the LinkBar to specify its label text. Use the following example code to create the LinkBar container shown in the previous figure:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

    <mx:LinkBar borderStyle="solid"
        click="getURL('http://www.macromedia.com/' +
            String(event.label).toLowerCase(), '_blank');" >
        <mx:dataProvider>
            <mx:Array>
                <mx:String>Flash</mx:String>
                <mx:String>Director</mx:String>
                <mx:String>Dreamweaver</mx:String>
                <mx:String>ColdFusion</mx:String>
            </mx:Array>
        </mx:dataProvider>
    </mx:LinkBar>
</mx:Application>
```

In this example, you use the `<mx:dataProvider>` and `<mx:Array>` tags to define the label text. The event object passed to the `click` handler contains the label selected by the user. The handler for the `click` event constructs an HTTP request to the Macromedia website based on the label, and opens that page in a new browser window.

You can also bind data to the `<mx:dataProvider>` tag to populate the LinkBar container, as the following example shows:

```
<mx:Script>
    <![CDATA[
        var linkData:Array = ["Flash", "Director", "Dreamweaver", "ColdFusion" ];
    ]]>
</mx:Script>

<mx:LinkBar horizontalAlign="right" borderStyle="solid"
    click="getURL('http://www.macromedia.com/' +
        String(event.label).toLowerCase(), '_blank');" >
    <mx:dataProvider>
        {linkData}
    </mx:dataProvider>
</mx:LinkBar>
```

In this example, you define the data for the LinkBar container as a variable in ActionScript, and then you bind that variable to the `<mx:dataProvider>` tag. You could also bind to the `<mx:dataProvider>` tag from a Flex data model, from a web service response, or from any other type of data model.

LinkBar container syntax

You use the `<mx:LinkBar>` tag to define a LinkBar container. The LinkBar container inherits all of the properties of the classes `MovieClip`, `UIObject`, `UIComponent`, `View`, and `Container`. For a list of these properties, see [“Configuring containers” on page 185](#). For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*.

This container also defines the following properties:

| Property | Type | Use | Description |
|---------------------------|-----------------|----------|--|
| <code>dataProvider</code> | Array or String | Property | Specifies the data used to populate the LinkBar navigator, or the identifier of a ViewStack container to associate with the LinkBar navigator. Flex automatically populates the LinkBar labels using the contents of the ViewStack container. If the child containers of the ViewStack container define an <code>icon</code> property, the icon appears in the corresponding link of the LinkBar container. |
| <code>labelField</code> | String | Property | Specifies the name of the field in the <code>dataProvider</code> Array to use as the label field. The label field defines the string that appears in each link. If the data provider is a ViewStack container, this property has no effect. If omitted, the Array must contain a field named <code>label</code> , or the <code>dataProvider</code> property must contain an Array of Strings. For an example using this property, see “Accordion navigator container” on page 263 . |
| <code>click</code> | | Event | • Broadcast when the user selects a label. |

TabBar navigator container

A TabBar navigator container defines a horizontal row of tabs. The following figure shows an example of a TabBar container:



Like the LinkBar container, you can use a TabBar navigator container to control the active child container of a ViewStack container. The syntax for using a TabBar container to control the active child of a ViewStack container is the same as for a LinkBar container. For an example, see [“ViewStack navigator container” on page 248](#).

While a TabBar container is similar to a TabNavigator container, it does not have any children. For example, you use the tabs of a TabNavigator container to select its visible child container. You can use a TabBar container to control the visible contents of a single container to make that container’s children visible or invisible based on the selected tab.

A TabBar container has the following default properties:

| Property | Default |
|--------------------------|---|
| Preferred size | A width wide enough to contain all label text, plus any margins, and a height tall enough for the label text. The default tab height is determined by the font, style, and skin applied to the container. If you set an explicit height using the <code>tabHeight</code> property, that value overrides the default value. |
| Container resizing rules | LinkBar containers do not resize by default. Specify percentage sizes if you want your LinkBar to resize based on the size of its parent container. |
| Margins | Left, right = 0 pixels. |

Creating a TabBar container

You use the `<mx:TabBar>` tag to define a TabBar container in MXML. Specify an `id` value if you intend to refer to a component elsewhere in your MXML, either in another tag or in an ActionScript block.

You specify the data for the TabBar container using the `<mx:dataProvider>` and `<mx:Array>` child tags of the `<mx:TabBar>` tag. The `<mx:dataProvider>` tag lets you specify data in several different ways. In the simplest case for creating a TabBar container, you use the `<mx:dataProvider>`, `<mx:Array>`, and `<mx:String>` tags to specify the text for each tab, as the following example shows:

```
<mx:TabBar>
  <mx:dataProvider>
    <mx:Array>
      <mx:String>Alabama</mx:String>
      <mx:String>Alaska</mx:String>
      <mx:String>Arkansas</mx:String>
    </mx:Array>
  </mx:dataProvider>
</mx:TabBar>
```

The `<mx:String>` tags define the text for each tab in the TabBar container.

You can also use the `<mx:Object>` tag to define the entries as an array of objects, where each object contains a `label` property and an associated data value, as the following example shows:

```
<mx:TabBar>
  <mx:dataProvider>
    <mx:Array>
      <mx:Object label="Alabama" data="Montgomery"/>
      <mx:Object label="Alaska" data="Juneau"/>
      <mx:Object label="Arkansas" data="Little Rock"/>
    </mx:Array>
  </mx:dataProvider>
</mx:TabBar>
```


The `label` property contains the state name and the `data` property contains the name of its capital. The `data` property lets you associate a data value with the text label. For example, the `label` text could be the name of a color, and the associated data value could be the numeric representation of that color.

By default, Flex uses the value of the `label` property to define the tab text. If the object does not contain a `label` property, you can use the `labelField` property of the `TabBar` container to specify the property name containing the tab text, as the following example shows:

```
<mx:TabBar labelField="state" >
  <mx:dataProvider>
    <mx:Array>
      <mx:Object state="Alabama" data="Montgomery"/>
      <mx:Object state="Alaska" data="Juneau"/>
      <mx:Object state="Arkansas" data="Little Rock"/>
    </mx:Array>
  </mx:dataProvider>
</mx:TabBar>
```

Passing data to a TabBar container

Flex lets you populate a `TabBar` container from an ActionScript variable definition or from a Flex data model. When you use a variable, you can define it to contain one of the following:

- A label (string)
- A label (string) paired with data (scalar value or object)

The following example populates a `TabBar` container from a variable:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

  <mx:Script>
    <![CDATA[
      var STATE_ARRAY:Array=
        [{label:"Alabama", data:"Montgomery"},
         {label:"Alaska", data:"Juneau"},
         {label:"Arkansas", data:"LittleRock"} ];
    ]]>
  </mx:Script>

  <mx:TabBar >
    <mx:dataProvider>
      {STATE_ARRAY}
    </mx:dataProvider>
  </mx:TabBar>

</mx:Application>
```

You can also bind a Flex data model to the `dataProvider` property. For more information on using data models, see [Chapter 28, “Managing Data in Flex,” on page 629](#).

Handling TabBar container events

The TabBar container defines a `click` event that is broadcast when a user selects a tab. The event object contains the following properties:

`label` String containing the label of the selected tab.

`index` Number containing the index of the selected tab. Indexes are in the range of 0, 1, 2, ..., *n* - 1, where *n* is the total number of tabs. The default value is 0, corresponding to the first tab.

The following example code shows a handler for the `click` event for this TabBar container:

```
<mx:Script>
  <![CDATA[

    var STATE_ARRAY:Array=
      [{label:"Alabama", data:"Montgomery"},
       {label:"Alaska", data:"Juneau"},
       {label:"Arkansas", data:"LittleRock"}
      ];

    function clickEvt(evt) {
      forClick.text="label is: " + evt.label + " index is: " + evt.index +
        " capital is: " + evt.target.dataProvider[evt.index].data;
    }
  ]]>
</mx:Script>

<mx:TabBar id="myTB" click="clickEvt(event)" >
  <mx:dataProvider>
    {STATE_ARRAY}
  </mx:dataProvider>
</mx:TabBar>

<mx:TextArea id="forClick" width="150" />
```

In this example, every `click` event updates the `TextArea` control with the tab label, selected index, and the selected data from the TabBar container's `dataProvider` Array.

TabBar container syntax

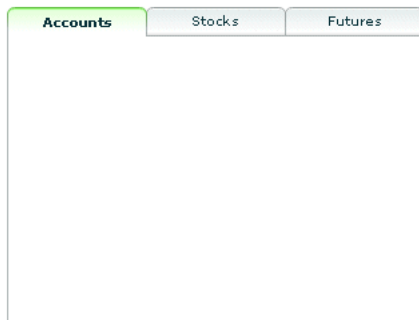
You use the `<mx:TabBar>` tag to define a TabBar container. The TabBar container inherits all of the properties of the classes `MovieClip`, `UIObject`, `UIComponent`, `View`, and `Container`. For a list of these properties, see [“Configuring containers” on page 185](#). For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*.

This container also defines the following properties:

| Property | Type | Use | Description |
|--|-----------------|----------|---|
| <code>activeTabStyleDeclaration</code> | String | Property | Identifies a type selector containing text styles for the currently selected tab. |
| <code>dataProvider</code> | Array or String | Property | Specifies the data used to populate the TabBar container, or the identifier of a ViewStack container to associate with the TabBar container. Flex automatically populates the TabBar tabs using the contents of the ViewStack container. If the child containers of the ViewStack container define an <code>icon</code> property, the icon appears in the corresponding tab of the TabBar container. |
| <code>labelField</code> | String | Property | Specifies the name of the property in the <code>dataProvider</code> Array to use as the label field. The label field defines the string that appears in each tab. If omitted, the Array must contain a field named <code>label</code> , or the <code>dataProvider</code> property must contain an Array of Strings. If the data provider is a ViewStack container, this property has no effect. |
| <code>selectedIndex</code> | Number | Property | Specifies the index of the active tab. Indexes are in the range of 0, 1, 2, ..., $n - 1$, where n is the total number of tabs. The default value is 0, corresponding to the first tab. |
| <code>click</code> | | Event | <ul style="list-style-type: none">• Broadcast when the user selects a tab. |

TabNavigator container

A TabNavigator container creates and manages a set of tabs, which you use to navigate among its children. The children of a TabNavigator container are other containers. The TabNavigator container creates one tab for each child. When the user selects a tab, the TabNavigator container displays the associated child, as the following figure shows:



The TabNavigator container is a child class of the ViewStack container and inherits much of its functionality. A TabNavigator container has the following default properties:

| Property | Default |
|--------------------------|---|
| Preferred size | The width and height of the first active child plus the tabs, or the width required to display all the tabs at their minimum size (30 pixels), whichever is larger. The default tab height is determined by the font, style, and skin applied to the TabNavigator container. If you set an explicit height using the <code>tabHeight</code> property, that value overrides the default value. |
| Container resizing rules | TabNavigator containers are only sized once to fit the size of the first child container by default. They do not resize when you navigate to other child containers by default. To force TabNavigator containers to resize when you navigate to a different child container, set the <code>resizeToContent</code> property to <code>true</code> . You can ensure that all of your child containers fit in the TabNavigator container with the expected layout by setting the <code>width</code> and <code>height</code> properties to explicit pixel values of sufficient size. You can also set both dimensions to 100% so that the container fills all available space. |
| Child sizing rules | Children are sized to their preferred size. The child is clipped if it is larger than the TabNavigator container. If the child is smaller than the TabNavigator container, it is aligned to the upper-left corner of the TabNavigator container. |
| Default margins | Top, bottom, left, right = 0 pixels. |
| Background color | White. |

Creating a TabNavigator container

Only one child of the TabNavigator container is visible at a time. Users can make any child the selected child by selecting its associated tab, or using keyboard navigation controls. Whenever the user changes the current child, the TabNavigator container broadcasts a `change` event.

The TabNavigator container automatically creates a tab for each of its children and determines the tab text from the `label` property of the child. The tabs are arranged left to right in the order determined by the child indexes. All tabs are visible, unless they do not fit within the width of the TabNavigator container.

The following code creates the TabNavigator container shown in the previous figure:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

  <mx:TabNavigator borderStyle="solid" >

    <mx:VBox label="Accounts" width="300" height="150" >
      <!-- Accounts view goes here. -->
    </mx:VBox>

    <mx:VBox label="Stocks" width="300" height="150" >
      <!-- Stocks view goes here. -->
    </mx:VBox>

    <mx:VBox label="Futures" width="300" height="150" >
```

```

        <!-- Futures view goes here. -->
    </mx:VBox>

</mx:TabNavigator>
</mx:Application>

```

You can also set the currently active child using the `selectedChild` and `selectedIndex` properties inherited from the `ViewStack` container as follows:

selectedIndex The index of the currently active container if one or more child containers are defined; undefined if no child containers are defined. The index is zero-based, 0, 1, 2, ... , $n - 1$, where n is the total number of child containers. Set this property to the index of the container that you want active.

selectedChild The identifier of the currently active container if one or more child containers are defined; undefined if no child containers are defined. Set this property to the identifier of the container that you want active. You can only set this property in an `ActionScript` statement, not in `MXML`.

For more information on the `selectedChild` and `selectedIndex` properties, including examples, see [“ViewStack navigator container” on page 248](#).

You use the `changeEffect` property to specify an effect to play when the user changes the currently active child. The default effect is `none`. The following example uses the `WipeLeft` effect for a `TabNavigator` container:

```
<mx:TabNavigator borderStyle="solid" changeEffect="WipeLeft" >
```

Sizing the children of a TabNavigator container

The default width and height of a `TabNavigator` container is the width and height of the first child. A `TabNavigator` container does not change size every time you change the active child.

You can use the following techniques to control the size of a `TabNavigator` container so that it displays all of the components inside its children:

- Set explicit `width` and `height` properties for all children to the same fixed values.
- Set the `width` and `height` properties for all children to the same percentage values.
- Set `width` and `height` properties for the `TabNavigator` container to 100%.

The method that you use is based on your application and the content of your `TabNavigator` container.

Order of initialization events

When you run your application, Flex broadcasts the `initialize` events for all top-level child containers of the `TabNavigator` container, and for the children of the initially visible child container. The order in which the `initialize` events occur is as follows:

1. For the initially visible child container
2. For the remaining child containers
3. For the children of the child containers as those containers become visible

You can also use the `childrenCreated` event with a `TabNavigator` container. This event is broadcast after a container creates its children. This event lets you perform one-time initialization of a container's children, just after they get created.

For example, a `TabNavigator` container creates all its immediate child containers, and all the children of its first visible child container. For the first visible child container, `childrenCreated` gets broadcast. Then, as the user moves to each additional child of the `TabNavigator` container, the event gets dispatched for that container.

Keyboard navigation

When a `TabNavigator` container has focus, Flex processes keystrokes as described in the following table:

| Key | Action |
|-------------------------|--|
| Down Arrow, Right Arrow | Gives focus to the next tab, wrapping from last to first, without changing the selected child. |
| Up Arrow, Left Arrow | Gives focus to the previous tab, wrapping from first to last, without changing the selected child. |
| Page Down | Selects the next child, wrapping from last to first. |
| Page Up | Selects the previous child, wrapping from first to last. |
| Home | Selects the first child. |
| End | Selects the last child. |
| Enter, Space | Selects the child associated with the tab displaying focus. |

TabNavigator container syntax

You use the `<mx:TabNavigator>` tag to define a `TabNavigator` container. The `TabNavigator` container inherits all of the properties of the classes `MovieClip`, `UIObject`, `UIComponent`, `View`, and `Container`. For a list of these properties, see [“Configuring containers” on page 185](#). For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*.

This container also defines the following properties:

| Property | Type | Use | Description |
|--|---------|----------|--|
| <code>activeTabStyleDeclaration</code> | String | Property | Identifies a type selector containing text styles for the currently selected tab. |
| <code>historyManagement</code> | Boolean | Property | Specifies whether to enable history management, <code>true</code> , or not, <code>false</code> . The default value is <code>true</code> . For more information, see Chapter 23, “Using the History Manager,” on page 569 . |

Accordion navigator container

Forms are a basic component of many applications. However, users have difficulty navigating through complex forms, or moving back and forth through multipage forms. Sometimes, forms can be so large that they do not fit onto a single screen.

Flex includes the Accordion navigator container, which can greatly improve the look and navigation of a form. The Accordion container defines a sequence of child panels, but displays only one panel at a time. The following figure shows an example of an Accordion container:

The diagram illustrates an Accordion navigator container with four panels:

- 1. Shipping Address**: This panel is currently expanded. It contains the following fields:
 - First Name:
 - Last Name:
 - Address:
 - City:
 - Phone:
 - State: (dropdown menu)
 - Zip Code:
 - Continue:
- 2. Billing Address**: Collapsed panel.
- 3. Credit Card Information**: Collapsed panel.
- 4. Submit Order**: Collapsed panel.

Navigation arrows are shown on the right side of the container:

- A single arrow points to the **1. Shipping Address** panel header.
- A bracketed arrow points to the **2. Billing Address**, **3. Credit Card Information**, and **4. Submit Order** panel headers.

To navigate a container, the user clicks on the navigation button that corresponds to the child panel that they want to access. Accordion containers let users access the child panels in any order to move back and forth through the form. For example, when the user is at the Credit Card Information panel, they might decide to change the information on the Billing Address panel. To do so, they navigate to the Billing Address panel, edit the information, and then navigate back to the Credit Card Information panel.

In HTML, a form that contains shipping address, billing address, and credit card information is often implemented as three separate pages, which requires the user to submit each page to the server before moving on to the next. An Accordion container can organize the information on three child panels with a single submit button. This architecture minimizes server traffic and lets the user maintain a better sense of progress and context.

Although Accordion containers are useful for working with forms, you can use any Flex component within a child panel of an Accordion. For example, you could create a catalog of products in an Accordion container, where each panel contains a group of similar products.

An Accordion container has the following default properties:

| Property | Default |
|--------------------------|---|
| Preferred size | The width and height of the currently active child. |
| Container resizing rules | Accordion containers are only sized once to fit the size of the first child container by default. They do not resize when you navigate to other child containers by default. To force Accordion containers to resize when you navigate to a different child container, set the <code>resizeToContent</code> property to <code>true</code> . You can ensure that all of your child containers fit in the Accordion container with the expected layout by setting the <code>width</code> and <code>height</code> properties to explicit pixel values of sufficient size. You can also set both dimensions to 100% so that the container fills all available space. |
| Child sizing rules | Children are sized to their preferred size. The child is clipped if it is larger than the Accordion container. If the child is smaller than the Accordion container, it is aligned to the upper-left corner of the Accordion container. |
| Default margins | Top, bottom, left, right = -1 pixel |

Creating an Accordion container

You define an Accordion container using the `<mx:Accordion>` tag, as the following example shows:

```
<mx:Accordion id="accordion1" height="450">
```

```
    <!-- Accordion definition.-->
```

```
</mx:Accordion>
```

Within the Accordion container, you define one container for each child panel. For example, if the Accordion container has four child panels that the correspond to four parts of a form, you define each child panel using the Form container, as the following example shows:

```
<?xml version="1.0"?>
```

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
```

```
    <mx:Accordion id="accordion1" height="450">
```

```
        <mx:Form id="shippingAddress" label="1. Shipping Address">
```

```
            <mx:FormItem id="sfirstNameItem" label="First Name">
```

```
                <mx:TextInput id="sfirstName" />
```

```
            </mx:FormItem>
```

```
            ...
```

```
        </mx:Form>
```

```
        <mx:Form id="billingAddress" label="2. Billing Address">
```

```
            ...
```

```
        </mx:Form>
```

```
        <mx:Form id="creditCardInfo" label="3. Credit Card Information">
```



```

    ...
</mx:Form>

<mx:Form id="submitOrder"    label="4. Submit Order">
    ...
</mx:Form>

</mx:Accordion>
</mx:Application>

```

This example defines each child panel using a Form container. However, you can use any container to define a child panel.

Note: All containers can be used to define child panels, however some containers do not really belong in them, such as a TabNavigator container or another Accordion container.

Keyboard navigation

When an Accordion container has focus, Flex processes keystrokes as described in the following table:

Note: An empty Accordion container with no child panels cannot take focus.

| Key | Action |
|-----------|---|
| Page Up | Move to the previous child panel, if any. |
| Page Down | Move to the next child panel, if any. |
| Home | Move to the first child panel. |
| End | Move to the last child panel. |

Using Button controls to navigate an Accordion container

The simplest way for users to navigate the panels of an Accordion container is to click the navigator button for the desired panel. However, many Accordion containers include additional navigation Button controls, such as Back and Next, to make it easier for users to navigate.

Navigation Button controls use the following properties of the Accordion container to move among the child panels:

`selectedIndex` The index of the currently active child panel. Child panels are numbered from 0 to $n-1$ where n is the total number of panels in the Accordion container. Writing to `selectedIndex` changes the currently active panel.

`selectedChild` The identifier of the currently active child container if one or more child containers are defined; undefined if no child containers are defined. Set this property to the identifier of the container that you want active. You can only set this property in an ActionScript statement, not in MXML.

`numChildren` Contains the total number of child panels defined in an Accordion container. The Accordion container inherits this property from the View class.

For more information on these properties, see [“ViewStack navigator container” on page 248](#).

For example, you can use the following two Button controls within the second panel of an Accordion container, panel number 1, to move back to panel number 0 or ahead to panel number 2:

```
<mx:Button id="backButton" label="Back" click="accordion1.selectedIndex=0;" />
<mx:Button id="nextButton" label="Next" click="accordion1.selectedIndex=2;" />
```

You can also use relative location with navigation buttons. The following Button controls move forward and back through Accordion container panels based on the current panel number:

```
<mx:Button id="backButton" label="Back"
    click="accordion1.selectedIndex = accordion1.selectedIndex - 1;" />
<mx:Button id="nextButton" label="Next"
    click="accordion1.selectedIndex = accordion1.selectedIndex + 1;" />
```

For the Next Button control, you also can use the `selectedChild` property to move to the next panel based on the `id` property of the panel's container, as the following code shows:

```
<mx:Button id="nextButton" label="Next"
    click="accordion1.selectedChild=creditCardInfo;" />
```

The following Button control opens the last panel in the Accordion container:

```
<mx:Button id="lastButton" label="Last"
    click="accordion1.selectedIndex = accordion1.numChildren - 1;" />
```

Handling child button events

The Accordion container can recognize an event when the user changes the current panel. The Accordion container broadcasts a `change` event when the user changes the child panel, either by clicking a button or pressing a key, such as the Page Down key.

Note: A `change` event is not broadcast when the child panel changes programmatically.

You can register an event handler for the `change` event using the `change` property of the `<mx:Accordion>` tag, or by registering the handler in ActionScript.

Order of initialization events

When you run your application, Flex broadcasts the `initialize` events for all top-level child containers of the Accordion container, and for the children of the initially visible child container. The order in which the `initialize` events occur is as follows:

1. For the initially visible child container
2. For the remaining child containers
3. For the children of the child containers as those containers become visible

You can also use the `childrenCreated` event with a Accordion container. This event is broadcast after a container creates its children. This event lets you perform one-time initialization of a container's children, just after they get created.

For example, an Accordion container creates all its immediate child containers, and all the children of its first visible child container. For the first visible child container, `childrenCreated` gets broadcast. Then, as the user moves to each additional child of the Accordion container, the event gets dispatched for that container.

Accordion container syntax

You use the `<mx:Accordion>` tag to define an Accordion container. The Accordion container inherits all of the properties of the classes `MovieClip`, `UIObject`, `UIComponent`, `View`, and `Container`. For a list of these properties, see [“Configuring containers” on page 185](#). For a complete description of the syntax, see *Flex ActionScript and MXML API Reference*.

This container also defines the following properties:

| Property | Type | Use | Description |
|--------------------------------|---------|----------|---|
| <code>historyManagement</code> | Boolean | Property | Specifies whether to enable history management, <code>true</code> , or not, <code>false</code> . The default value is <code>true</code> . For more information, see Chapter 23, “Using the History Manager,” on page 569 . |
| <code>resizeToContent</code> | Boolean | Property | Specifies whether the ViewStack container resizes when you navigate to a different child container. The default value is <code>false</code> . |
| <code>selectedChild</code> | String | Property | Specifies the name of the active container. The default is the identifier of the first child container defined in the Accordion container. You can only set this property in an ActionScript statement, not in MXML. |
| <code>selectedIndex</code> | Number | Property | Specifies the index of the active container when the Accordion container loads. Indexes are in the range of 0, 1, 2, ..., $n - 1$, where n is the total number of child containers. The default value is 0, corresponding to the first child container defined in the Accordion container. |
| <code>change</code> | | Event | <ul style="list-style-type: none">• Broadcast when the current view changes. |

CHAPTER 8

Dynamically Repeating Controls and Containers

This chapter describes how to use the Repeater object. The Repeater object lets you dynamically repeat any number of controls or containers specified in MXML tags, at runtime.

The Repeater object is useful for repeating a small set of simple user-interface components, such as RadioButton controls and other controls typically used in Form containers. A component is repeated based on an array of dynamic data, such as an Array object returned from a web service. For example, using just one `<mx:RadioButton>` tag and one `<mx:Repeater>` tag, you can generate a RadioButton control for each element in an Array object.

For most cases in which you would use the Repeater object in Macromedia Flex 1.0, you can now use the HorizontalList, TileList, or List control for better performance when displaying any more than a few objects. Unlike the Repeater object, which instantiates all objects that are repeated, the HorizontalList, TileList, and List controls only instantiate what is visible in the list.

The HorizontalList control is a list control that displays data horizontally, much like the HBox container. The HorizontalList control always displays items from left to right. For more information, see [“HorizontalList control” on page 116](#).

The TileList control is a list control that displays data in a tile layout, much like the Tile container. The TileList control provides a `direction` property that determines if the next item is down or to the right. For more information, see [“TileList control” on page 121](#).

The List control displays data in a single vertical column. For more information, see [“List control” on page 105](#).

Contents

| | |
|---|-----|
| Using the Repeater object | 270 |
| Considerations when using a Repeater object | 279 |

Using the Repeater object

You use the `<mx:Repeater>` tag to declare a Repeater object that handles repetition of one or more user-interface components based on dynamic data arrays at runtime. The repeated components can be controls or containers. Using a Repeater object requires data binding; for more information about data binding, see [Chapter 29, “Binding and Storing Data in Flex,”](#) on [page 637](#).

You can use the `<mx:Repeater>` tag anywhere a control or container tag is allowed, with the exception of the `<mx:Application>` container tag. To repeat user interface components, you place their tags in the `<mx:Repeater>` tag. You can also use more than one `<mx:Repeater>` tag in an MXML document, and you can nest `<mx:Repeater>` tags.

You can use the `<mx:Repeater>` tag only for objects that extend the `UIObject` class.

Declaring a Repeater object in MXML

You declare a Repeater object in the `<mx:Repeater>` tag. The following table describes the Repeater object properties:

| Property | Description |
|---------------------------|---|
| <code>id</code> | Instance name of the corresponding Repeater object. |
| <code>dataProvider</code> | <p>Array object or an object that supports the <code>length</code> property and <code>getItemAt()</code> method of the <code>mx.controls.listclasses.DataProvider</code> class. You must specify a <code>dataProvider</code> property to repeat components.</p> <p>Generally, you specify the value of the <code>dataProvider</code> property as a binding expression because the value is not known until runtime. The following example shows a <code>dataProvider</code> property bound to a web service result:</p> <pre>... <mx:Repeater id="r" dataProvider="{ws.getNames.result}"> <!-- User interface component tag goes here. --> </mx:Repeater></pre> |
| <code>startIndex</code> | Number that specifies the zero-based array index at which the repetition starts. If the <code>startIndex</code> is not within the range of the <code>dataProvider</code> property, no repetition occurs. |
| <code>count</code> | Number that specifies how many repetitions occur. If there are fewer items in the <code>dataProvider</code> property, the repetition stops with the last item. |
| <code>currentIndex</code> | Number that is the zero-based index of the <code>dataProvider</code> item that is being processed. This property changes as the Repeater object executes, and is undefined after the execution is complete. It is a read-only property that you cannot set in the <code><mx:Repeater></code> tag. |

| Property | Description |
|------------------------------|---|
| <code>currentItem</code> | Reference to the item that is being processed in the <code>dataProvider</code> property. This property changes as the Repeater object executes, and is undefined after the execution is complete. It is a read-only property that you cannot set in the <code><mx:Repeater></code> tag. |
| <code>recycleChildren</code> | <p>Boolean value that, when set to <code>true</code>, binds new data items into existing Repeater children, incrementally creates new children if there are more data items, and destroys extra children that are no longer required.</p> <p>When you set this property to <code>false</code>, the Repeater object recreates all the objects when you swap one <code>dataProvider</code> with another, sort, and so on, which causes a performance lag. Only set this property to <code>false</code> if you are confident that modifying your <code>dataProvider</code> should not recreate the Repeater object's children.</p> <p>The default value of this property is <code>false</code>, to ensure that you do not leave stale state information in a repeated instance. For example, suppose you use a Repeater object to display photo images, and each Image control has an associated NumericStepper control for how many prints you want to order. Some of the state information, the image, comes from the <code>dataProvider</code>, while other state information, the print count, is set by user interaction. If you set the <code>recycleChildren</code> property to <code>true</code> and page through the photos by incrementally increasing the Repeater object's <code>startIndex</code> value, the Image controls bind to the new images, but the NumericStepper controls keep the old information.</p> |

To allow for event handling, the Repeater object dispatches a `repeat` event each time an item is processed and the `currentIndex` and `currentItem` properties are updated.

In the following example, an `<mx:Repeater>` tag repeats a `RadioButton` control for each product in a data model named *catalog*:

```
<mx:Model id="catalog" source="catalog.xml"/>
...
<mx:Repeater dataProvider="{catalog.product}">
    <mx:RadioButton id="Radio" label="product.name"/>
</mx:Repeater>
```

Referencing repeated components in ActionScript

To reference individual instances of a repeated component in ActionScript, you use indexed `id` references on the document object, as the following example shows:

```
<mx:Label id="title" text="Employees:"/>
<mx:Repeater dataProvider="{ws.getNames.result}">
    <mx:Label id="nameLabel"
        text="{r.currentItem.firstName} {r.currentItem.lastName}"/>
</mx:Repeater>
...
<mx:Script>
    <![CDATA[
        labelTrace(){
            for (var i = 0; i < nameLabel.length; i++)
                trace(nameLabel[i].text);
        }
    ]]>
</mx:Script>
```

In this example, the `id` of the repeated `Label` control is `nameLabel`; each `nameLabel` instance that is created has this `id`. You reference the individual `Label` instances as `nameLabel[0]`, `nameLabel[1]`, and so on. You reference the total number of `nameLabel` instances as `nameLabel.length`. The `for` loop traces the `text` property of each `Label` control in the `nameLabel` Array object.

Referencing repeated child components

When a container is repeated and indexed in an Array object, its children are also indexed. For example, for the following MXML code, you reference the child `Label` controls of the `HBox` container `hb[0]` as `nameLabel[0]` and `locationLabel[0]`. The syntax for referencing the children is the same as the syntax for referencing the parent.

```
<mx:Label id="title">Employees:</mx:Label>
<mx:Repeater dataProvider="{ws.getNames.result}">
  <mx:HBox id="hb">
    <mx:Label id="nameLabel"
      text="{r.currentItem.firstName} {r.currentItem.lastName}"/>
    <mx:Label id="locationLabel"
      text="{r.currentItem.city}, {r.currentItem.state}"/>
  </mx:HBox>
</mx:Repeater>
```

Referencing repeated child Repeater objects

When `<mx:Repeater>` tags are nested, the inner `<mx:Repeater>` tags are indexed `Repeater` objects. For example, for the following MXML code, you access the nested `Repeater` objects as `r2[0]`, `r2[1]`, and so on. The syntax for referencing the children is same as the syntax for referencing the parent.

```
<mx:Repeater id="r1" dataProvider="{...}">
  <mx:Repeater id="r2" dataProvider="{...}">
    <mx:RadioButton id="Radio"/>
  </Repeater>
</mx:Repeater>
```

For the nested `Repeater` objects in the previous example, the instances of the `Button` control are multiple-indexed because they are inside multiple `Repeater` objects. For example, the index `b[2][4]` contains a reference to the `Button` control produced by the third iteration of `r1` and the fifth iteration of `r2`.

When a `Repeater` object is busy repeating, each repeated object that it creates can bind at that moment to the `Repeater` object's `currentItem` property, which is changing as the `Repeater` object repeats. You cannot give each instance its own event handler by writing something like `click="doSomething({r.currentItem})"` because binding expressions are not allowed in event handlers, and all instances of the repeated component must share the same event handler.

Repeated components and repeated Repeater objects have a `getRepeaterItem()` method that returns the item in the `dataProvider` property that was used to produce the object. When the Repeater object finishes repeating, you can use the `getRepeaterItem()` method to determine what the event handler should do based on the `currentItem` property. To do so, you pass `event.target.getRepeaterItem()` to the event handler, as the following example shows. When the user clicks each repeated Button control, the corresponding `colorName` value from the data model is displayed in the Button control label:

```
<?xml version="1.0"?>
<mx:Application borderStyle="solid" height="550" width="750"
xmlns:mx="http://www.macromedia.com/2003/mxml" >

  <mx:Label id="foolabel" text="foo"></mx:Label>

  <mx:Model id="data">
    <colorName>Red</colorName>
    <colorName>Yellow</colorName>
    <colorName>Blue</colorName>
  </mx:Model>

  <mx:Script>
    <![CDATA[
      function clicker(cName)
      {
        foolabel.text=cName;
      }
    ]]>
  </mx:Script>

  <mx:Repeater id="myrep" dataProvider="{data.colorName}">
    <mx:Button click="clicker(event.target.getRepeaterItem());"
      label="{myrep.currentItem}"/>
  </mx:Repeater>
</mx:Application>
```

The `getRepeaterItem()` method takes an optional index that specifies which Repeater object you want when there are nested Repeater objects; the 0 index is the outermost Repeater object. If you do not specify the index argument, the innermost Repeater object is implied.

The code in the following example uses the `getRepeaterItem()` method to display a specific URL for each Button control that the user clicks. The Button controls must share a common data-driven click handler, because you cannot use binding expressions inside event handlers, but the `getRepeaterItem()` method lets you change the functionality for each Button control.

```
<Script>
  <![CDATA[
    var dp = [ { label: "Flex", url: "http://www.macromedia.com/flex" },
      { label: "Flash", url: "http://www.macromedia.com/flash" } ];
  ]]>
</Script>

<Repeater id="r" dataProvider="{dp}">
```

```

        <Button label="{r.currentItem.label}"
            click="displayUrl(event.target.getRepeaterItem().url)"/>
    </Repeater>

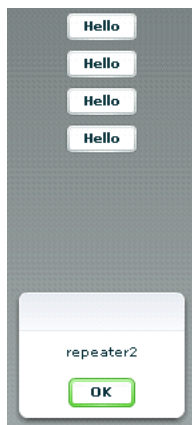
```

Accessing specific instances of repeated objects

Repeated components and repeated Repeater objects have three properties, `instanceIndices`, `repeaters`, and `repeaterIndices`, that you can use to dynamically keep track of specific instances of repeated objects, determine which Repeater object produced them, and determine which `dataProvider` items were used by each Repeater object. The following table describes these properties:

| Property | Description |
|------------------------------|--|
| <code>instanceIndices</code> | Array that contains the indices required to reference the object from its document. This Array is empty unless the object is in one or more Repeater objects. The first element corresponds to the outermost Repeater object. For example, if the <code>id</code> is <code>b</code> and <code>instanceIndices</code> is <code>[2,4]</code> , you would reference it on the document as <code>b[2][4]</code> . |
| <code>repeaters</code> | Array that contains references to the Repeater objects that produced the object. The Array is empty unless the object is in one or more Repeater objects. The first element corresponds to the outermost Repeater object. |
| <code>repeaterIndices</code> | Array that contains the indices of the items in the <code>dataProvider</code> properties of the Repeater objects that produced the object. The Array is empty unless the object is within one or more Repeater objects. The first element corresponds to the outermost Repeater object. For example, if <code>repeaterIndices</code> is <code>[2,4]</code> , the outer Repeater object used its <code>dataProvider[2]</code> data item and the inner Repeater object used its <code>dataProvider[4]</code> data item. This property differs from <code>instanceIndices</code> if the <code>startIndex</code> of any of the Repeater objects is not 0. For example, even if a Repeater object starts at <code>dataProvider</code> item 4, the document reference of the first repeated object is <code>b[0]</code> , not <code>b[4]</code> . |

The following example application uses the `repeaters` property to display the `id` value of the innermost repeater in an Alert control when the user clicks one of the Button controls:

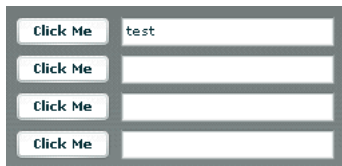


The code in the following example shows how to use the `repeaters` property to get the `id` value of the one nested Repeater object by using `repeaters[1].id`. To get the `id` value of the outermost repeater, you would use `repeaters[0].id`.

```
<?xml version="1.0"?>

<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" xmlns="*">
  <mx:Script>
    <![CDATA[
      var myArray=[1,2];
    ]]>
  </mx:Script>
  <mx:Repeater id="r1" dataProvider="{myArray}">
    <mx:Repeater id="repeater2" dataProvider="{myArray}">
      <mx:Button label="Hello" click="alert(event.target.repeaters[1].id)"/>
    </mx:Repeater>
  </mx:Repeater>
</mx:Application>
```

The following example application uses the `instanceIndices` property to set the `text` property of a `TextInput` control when the user clicks the corresponding `Button` control in the set of repeated `Button` and `TextInput` controls. It is necessary to use the `instanceIndices` property, because you must get the correct object dynamically; you cannot get it by its `id` value.



The code in the following example shows how to use the `instanceIndices` property to set the `text` property of a `TextInput` control when the user clicks a `Button` control. The argument `event.target.instanceIndices` gets the index of the corresponding `TextInput` control.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" xmlns="*">
  <mx:Script>
    <![CDATA[
      var myArray=[1,2,3,4,5,6,7,8];
    ]]>
  </mx:Script>

  <mx:Repeater id="list" dataProvider="{myArray}">
    <mx:HBox>
      <mx:Button label="Click Me"
        click="myText[event.target.instanceIndices].text='test'"/>
      <mx:TextInput id="myText"/>
    </mx:HBox>
  </mx:Repeater>
</mx:Application>
```

The code in the following example shows how to use the `repeaterIndices` property instead of the `instanceIndices` property to set the `text` property of a `TextInput` control when the user clicks a `Button` control. The value of `event.target.repeaterIndices` is based on the current index of the `Repeater` object. Because the `startIndex` property of the `Repeater` object is set to 2, it does not match the `event.target.instanceIndices` value.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" xmlns="*">
  <mx:Script>
    <![CDATA[
      var myArray=[1,2,3,4,5,6,7,8];
    ]]>
  </mx:Script>

  <mx:Repeater id="list" dataProvider="{myArray}" startIndex="2">
    <mx:HBox>
      <mx:Button id="b" label="Click Me"
        click="myText[event.target.repeaterIndices[0]].text='test'"/>
      <mx:TextInput id="myText"/>
    </mx:HBox>
  </mx:Repeater>
</mx:Application>
```

Using a Repeater object in a custom MXML component

You can use the `<mx:Repeater>` tag in an MXML component definition in the same way that you use it in an application file. When you use the MXML component as a tag in another MXML file, the repeated items appear. You can access an individual repeated item by its Array index number, just as you do for a repeated item defined in the application file.

In the following example, a `Button` control in an MXML component called `childComp` is repeated for every element in an Array object called *dp*:

```
<?xml version="1.0"?>
<mx:VBox xmlns:mx="http://www.macromedia.com/2003/mxml" width="100"
  height="100">

  <mx:Script>
    <![CDATA[
      var dp=[1,2,3,4];
    ]]>
  </mx:Script>

  <mx:Repeater id="r" dataProvider="{dp}">
    <mx:Button id="repbutton" label="button {r.currentItem}"/>
  </mx:Repeater>

</mx:VBox>
```

The application file in the following example uses the `childComp` component to display four Buttons, one for each element in the Array object. The `getLabelRep()` function displays the label text of the second Button in the Array object.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" width="500"
  height="500" >

  <mx:Script>
  <![CDATA[
    function getLabelRep()
    {
      alert(comp.repbutton[1].label)
    }
  ]]>
</mx:Script>

  <childComp id="comp"/>

  <mx:Button label="Get label of Repeated element" width="200"
    click="getLabelRep()"/>

</mx:Application>
```

Dynamically creating components based on data type

You can use a Repeater object to dynamically create different types of components for specific items in a set of data. A Repeater object broadcasts a `repeat` event as it executes, and this event is broadcast after you set the `currentIndex` and `currentItem` properties. You can call an event handler function on the `repeat` event, and dynamically create different types of components based on the individual data items.

In the following example, a Repeater reference is passed as `event.target` to a function called `isArray()`. The `isArray()` function is called for every item in the Repeater object's `dataProvider` property. The `isArray()` function checks whether the Repeater object's `currentItem` property is an Array. If the `currentItem` property is an Array, the function creates a `ComboBox` control using the `createChild()` method. If the `currentItem` is a String, the function creates a `TextInput` control.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:Script>
  <![CDATA[
    function getData1():Array {
      var r:Array = new Array();
      var vA:Array = new Array();
      vA.push("a");
      vA.push("b");
      var vB:Array = new Array();
      vB.push("c");
      vB.push("d");
      r.push("text A")
      r.push(vA);
    }
  ]]>
</mx:Script>
</mx:Application>
```

```

        r.push("text B")
        r.push(vB);
        return(r)
    }

    function isArray(myVal)
    {
        if(myVal.currentItem instanceof Array){
            b1.createChild(mx.controls.ComboBox, undefined,
                {dataProvider:myVal.currentItem})
        } else{
            b1.createChild(mx.controls.TextInput, undefined,
                {text:myVal.currentItem});
        }
    }
}]]>
</mx:Script>

<mx:Box id="b1" direction="vertical" borderStyle="solid" marginTop="10"
marginBottom="10" marginLeft="10" marginRight="10" >
    <mx:Repeater id="r1" dataProvider="{getData1()}"
        repeat="isArray(event.target)"/>
</mx:Box>

</mx:Application>

```

How a Repeater object executes

A Repeater object executes initially when it is instantiated. If the Repeater object's `dataProvider` property exists, it proceeds to instantiate its children, and they instantiate their children, recursively.

The Repeater object re-executes whenever its `dataProvider`, `startIndex`, or `count` properties are set either explicitly in `ActionScript` or implicitly by data binding. If the `dataProvider` property is bound to a web service result, the Repeater object re-executes when the web service operation returns the result. A Repeater object also re-executes in response to paging through the `dataProvider` property by incrementally increasing the `startIndex` value, as the following example shows:

```
r.startIndex += r.count;
```

When a Repeater object re-executes, it destroys any children that it previously created, and then reinstantiates its children based on the current `dataProvider` property. The number of children in the container might change, and the container layout changes to accommodate any changes to the number of children.

A Repeater object re-executes if you set its entire `dataProvider` value. However, a Repeater object does not re-execute if you set just one item of its `dataProvider` value. For example, a Repeater object does not re-execute for the following code:

```
r.dataProvider[3] = { imageUrl: "c740.jpg", modelName: "Olympus C-740 Ultra
Zoom " };
```

Instead, you can call the `dataProvider.replaceItemAt()` method, as the following example shows:

```
r.dataProvider.replaceItemAt(3, { imageUrl: "c740.jpg", modelName: "Olympus C-740 Ultra Zoom " });
```

When a property of a `dataProvider` item changes, a `Repeater` object does not re-execute, but it does update repeated component bindings to the item's property. In the following example, a `Repeater` object updates repeated bindings to `{r.currentItem.modelName}`:

```
r.dataProvider[3].modelName = "Olympus C-740 Ultra Zoom - BUY ME NOW";
```

Considerations when using a Repeater object

Consider the following when you use a `Repeater` object:

- You cannot use a `Repeater` object to iterate through a two-dimensional Array object that is programmatically generated. This is because the elements of an Array object do not trigger `changeEvent` events, and therefore cannot function as binding sources at runtime. Binding copies initial values during instantiation after variables are declared in an `<mx:Script>` tag, but before `initialize` handlers are executed.
- A `Repeater` object does not successfully create repeated objects if the `id` property of the object to be repeated is the same value as the `id` property of a dynamically created object. In the following example, a `Button` control with an `id` value of `button` is dynamically created in the `create_child()` function; this `Button` is successfully repeated in the `r1` `Repeater` object. A `Button` control with an `id` value of `button` is also created in an `<mx:Button>` tag in the `r2` `Repeater` object. The `Button` control in the `<mx:Button>` tag is not repeated, because it has the same `id` value as the dynamically created `Button` control.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:Script>
    <![CDATA[
      var dp = ['a','b', 'c']
      var dp1 = ['a1','b1', 'c1']

      function create_child()
      {
        bl.createChild(mx.controls.Button, "", {id:"button",
          label:"Button0"});
      }
    ]]>
  </mx:Script>

  <mx:VBox id="b1" direction="vertical" borderStyle="solid" marginTop="10"
    marginBottom="10" marginLeft="10" marginRight="10" width="100%"
    height="100%">

    <!-- This Repeater works as expected because it repeats the Button control
      that is dynamically created in the create_child() function. -->
    <mx:Repeater id="r1" dataProvider="{dp}" repeat="create_child()"/>

    <mx:Repeater id="r2" dataProvider="{dp1}" >
```

```

        <!-- This repeated Button is not created. -->
        <mx:Button id="button" label="Button1"/>

    </mx:Repeater>

</mx:VBox>
</mx:Application>

```

- Forgetting curly braces ({ }) in a `dataProvider` property is a common mistake when using a Repeater object. If the Repeater object doesn't execute, make sure that the binding is correct.
- If repeated objects are displayed out of order and you are using adjacent or nested Repeater objects, you might need to place a dummy UIObject immediately after the Repeater that is displaying objects incorrectly.

The code in the following example contains adjacent `<mx:Repeater>` tags and uses an `<mx:Spacer>` tag to create a dummy UIObject:

```

<mx:VBox>
    <mx:Repeater id="r1">
        ...
    </mx:Repeater>
    <mx:Repeater id="r2">
        ...
    </mx:Repeater>
    <mx:Spacer height="0" id="dummy" />
</mx:VBox>

```

The code in the following example contains nested `<mx:Repeater>` tags and uses an `<mx:Spacer>` tag to create a dummy UIObject:

```

<mx:VBox>
    <mx:Repeater id="outer">
        <mx:Repeater id="inner">
            ...
        </mx:Repeater>
        <mx:Spacer id="dummy" height="0">
    </mx:Repeater>
</mx:VBox>

```


CHAPTER 9

Importing Images

Macromedia Flex supports several image formats, including JPEG, PNG, GIF, and SVG images and SWF files. This chapter describes how to import these image types into a Flex application.

Contents

| | |
|---|-----|
| About importing images | 281 |
| Importing images using the <mx:Image> tag | 282 |
| Controlling image importing with the <mx:Image> tag | 284 |
| Referencing images in other MXML tags | 292 |
| Importing images using Embed in ActionScript | 293 |

About importing images

Flex supports both importing JPEG and SWF files at runtime and embedding JPEG, GIF, PNG, SVG, and SWF files into SWF files generated at compile time. The method you choose depends on the file types of your images and your application parameters.

Embedded images load immediately, because they are already part of the Flex SWF file. However, they add to the size of your application and slow down the application initialization process. Embedded images also require you to recompile your applications whenever your image files change. In addition, you cannot use embedded images in data binding calls, because data binding is a runtime operation. You can embed JPEG, PNG, GIF, SVG, and SWF files in a Flex application.

Referenced images exist as separate independent files on your web server (or elsewhere) and are not compiled into your Flex applications. The referenced images add no additional overhead to an application's initial load time. However, you might experience a delay when you actually use the image and load it into the Macromedia Flash Player. These images are independent of your Flex application, so you can change them without causing a recompile operation as long as the names of the modified images remain the same. Because the image is imported at runtime, you can use data binding to import referenced images and tie them to specific properties, as appropriate. You can only import JPEG and SWF files into your applications at runtime because these are the only file formats that are directly supported by Flash Player. Flash Player cannot load GIF, PNG, or SVG images at runtime.

Importing images using the `<mx:Image>` tag

Many Flex applications import stored images. For example, before buying a product, users typically want to see a picture of it in the product catalog.

Flex supports importing several of the most common image types, including JPEG, SVG, PNG, and GIF images, and SWF files. Flex also supports transparent GIF and PNG images.

Properties of the `<mx:Image>` tag

To import JPEG, PNG, GIF, and SVG images, and SWF files, you use the `<mx:Image>` tag. The `<mx:Image>` tag accepts the properties defined by the `UIObject` and `UIComponent` classes, and the properties described in the following table:

| Property | Type | Use | Description |
|----------------------------------|---------|----------|---|
| <code>source</code> | String | Property | (Required) Specifies the location of the imported file. |
| <code>scaleContent</code> | Boolean | Property | Specifies whether to scale the imported image to fit the size of this control, <code>true</code> , or not, <code>false</code> . The default value is <code>true</code> . |
| <code>maintainAspectRatio</code> | Boolean | Property | Specifies whether to maintain the aspect ratio of the imported image when resizing it, <code>true</code> , or not, <code>false</code> . The default value is <code>true</code> . |
| <code>complete</code> | | Event | Specifies a handler for <code>complete</code> events, which are broadcast when image loading completes. This event is only broadcast for images loaded at runtime. The event object contains a <code>target</code> property that contains a reference to the <code>Image</code> control that triggered the event. |
| <code>progress</code> | | Event | Specifies a handler for an event triggered while content is loading. This event is only broadcast for images loaded at runtime. The event is not guaranteed to be broadcast, meaning that the <code>complete</code> event might be received, without any <code>progress</code> events being broadcast. |

The value of the `source` property specifies a relative or absolute URL to the imported image file. If the URL is relative, it is relative to the directory that contains the file that uses the tag. For more examples, see [“Specifying the image path” on page 285](#).

The `source` property has the following forms:

`source="@Embed('relativeOrAbsoluteURL')"` The referenced image is packaged within the generated SWF file at compile-time when Flex creates the SWF file for your application.

For more information on embedding SWF files, see [“Importing SWF files” on page 283](#). For more information on embedding SVG images, see [“Importing SVG images” on page 284](#).

`source="relativeOrAbsoluteURL"` Flex loads the referenced image file at runtime; it is not packaged as part of the generated SWF file. You can only reference JPEG or SWF files.

The following example imports a JPEG image into a Flex application:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:VBox id="myVBox" width="500" height="500">
    <mx:Image id="image0" source="myJpeg.jpg" />
  </mx:VBox>
</mx:Application>
```

In this example, you include the `<mx:Image>` tag in a `VBox` (vertical box) container. The `VBox` container controls the location of the image. The size of the image is the default size of the image file. Because you did not use `@Embed` in the `source` property, Flex loads the image at runtime.

Using an image multiple times

You can use the same image multiple times in your application by using the normal image import syntax each time. Flex only loads the image once, and then references the loaded image as many times as necessary.

Importing SWF files

The `<mx:Image>` tag works primarily with small SWF files that add graphics or animations to an application; these files are not intended for user interaction. If you want to import SWF files that are built as Flex applications, or SWF files that require user interaction, you should build them as custom Flex components or as custom components in Flash MX 2004.

Restrictions on embedding SWF files

If you specify a SWF file as the value to the `source` property using `@Embed`, the embedded SWF file cannot contain any ActionScript 2.0 class definitions or Macromedia components. If it does, Flex does not embed the SWF file. If you must use a SWF file with ActionScript 2.0 class definitions or Macromedia components, load the SWF file at runtime.

Importing SWF file symbols

Flex lets you reference exported symbols in an imported SWF file. To reference the symbol, you specify it as part of the value to the `source` property using `@Embed`, in the following form:

```
@Embed('SWFFileName.swf#symbolName')
```

where the substring before the number sign (`#`) specifies the location of the SWF file, and the substring following the number sign (`#`) references an exported symbol in the SWF file.

This capability is useful when you have a SWF file that contains multiple exported symbols, but you only want to load some of them into your Flex application. Loading only the symbols required by your application makes your resulting Flex SWF file smaller than if you imported the entire SWF file.

Note: If you are only importing a symbol from a SWF file, the SWF file can contain ActionScript 2.0 classes and Macromedia components.

The following example imports a green square from a SWF file that contains a library of different shapes:

```
<mx:Image source="@Embed('shapes.swf#greenSquare')"/> />
```

Restriction on symbol access when importing SWF files

A Flex application can import any number of SWF files. However, if two SWF files have the same filename and duplicate exported symbol names, you cannot reference the duplicate symbols, even if the SWF files are in separate directories.

Importing SVG images

Flex supports importing Scalable Vector Graphics (SVG) images, or a GZip compressed SVG image in a SVGZ file, into an application. This lets you import SVG images using the `<mx:Image>` tag, and use SVG images as icons for Flex controls. For more information on the syntax of the `<mx:Image>` tag, see [“Properties of the `<mx:Image>` tag” on page 282](#).

Flex supports a subset of the SVG 1.1 specification to let you import static, 2-dimensional scalable vector graphics. This includes support for basic SVG document structure, Cascading Style Sheets (CSS) styling, transformations, paths, basic shapes, and colors, and a subset of text, painting, gradients, and fonts. Flex does not support SVG animation, scripting, or interactivity with the imported SVG image.

Controlling image importing with the `<mx:Image>` tag

The `<mx:Image>` tag supports the following actions when you import an image:

- [Specifying the image path](#)
- [Sizing an image](#)
- [Positioning the image in a Canvas container](#)
- [Setting visibility](#)
- [Using the Loader control](#)

Specifying the image path

In many applications, you create a directory to hold your application images. Commonly, that directory is a subdirectory of your main application directory. The `source` property supports relative paths to images, which lets you specify the location of an image file relative to your application directory.

The following example stores all images in an `images` subdirectory of the application directory:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:VBox id="vbox0" width="500" height="500">
    <mx:Image id="image0" source="images/myImage.jpg" />
  </mx:VBox>
</mx:Application>
```

The following example uses a relative path to reference an image in a directory above the application's root directory:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:VBox id="vbox0" width="500" height="500">
    <mx:Image id="image0" source="../../images/myImage.jpg" />
  </mx:VBox>
</mx:Application>
```

You can also reference an image using a URL, but the default security settings only permit Flex applications to access resources stored on the same domain as your application. Before you can use other URLs, you must explicitly add them as permitted resources. For more information, see [“Web services and HTTP services” on page 824](#).

The following example shows how to reference an image using a URL:

```
<mx:VBox id="vbox0" width="500" height="500">
  <mx:Image id="image0" source="http://myhost/images/myImage.jpg" />
</mx:VBox>
```

Note: You can use relative URLs for images hosted on the same web server as the Flex application, but these images are still be loaded over the Internet rather than accessed locally

You can also embed an image using a URL, as the following example shows:

```
<mx:VBox id="vbox0" width="500" height="500">
  <mx:Image id="image0" source="@Embed('http://myhost/images/myImage.gif')" />
</mx:VBox>
```

This lets you use GIF or PNG files that cannot be accessed at runtime. However, be aware that Flex does not monitor these remote images and a recompile operation will not be triggered automatically if they change.

Sizing an image

Flex sets the height and width of an imported image to the height and width settings in the image file. By default, Flex does not resize the image.

To set an explicit height or width for an imported image, set its `height` and `width` properties. Setting the `height` or `width` property prevents the parent from resizing it. Since the `scaleContent` property has a default value of `true`, Flex scales the image as it resizes it to fit the specified height and width. Set the `scaleContent` property to `false` to disable scaling.

To let Flex resize the image as part of laying out your application, set the `height` or `width` properties to a percentage value. Flex attempts to resize components with percentage values for these properties to the specified percentage of their parent container. You can also use the `maxHeight` and `maxWidth` and `minHeight` and `minWidth` properties to limit resizing. For more information on resizing, see [Chapter 4, “Introducing Containers,”](#) on page 161.

One common use for resizing an image is to create image thumbnails. In the following example, the image has an original height and width of 100 x 100 pixels. By specifying a height and width of 20 x 20 pixels, you create a thumbnail of the image.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
    <mx:VBox id="vbox0" >
        <mx:Image source="myImage.jpg" height="20" width="20" />
    </mx:VBox>
</mx:Application>
```

Maintaining aspect ratio when sizing

The aspect ratio of an image is the ratio of its width to its height. For example, a standard NTSC television set uses an aspect ratio of 4:3, while an HDTV set uses an aspect ratio of 16:9. A computer monitor with a resolution of 640 x 480 pixels also has an aspect ratio of 4:3. A square has an aspect ratio of 1:1.

All images have an inherent aspect ratio. When you use the `height` and `width` properties to resize an image, by default Flex preserves the aspect ratio of the image so that it does not appear distorted.

By preserving the aspect ratio of the image, Flex might not draw the image to fill the entire height and width specified for the `<mx:Image>` tag. For example, if your original image is a square 100 x 100 pixels, which means it has an aspect ratio of 1:1, and you use the following statement to load the image:

```
<mx:Image source="myImage.jpg" height="200" width="200" />
```

The image scales to four times its original size and fills the entire 200 x 200 pixel area.

The following example sets the height and width of the same image to 150 x 200 pixels, an aspect ratio of 3:4:

```
<mx:Image source="myImage.jpg" height="150" width="200" />
```

In this example, you do not specify a square area for the resized image. Since Flex maintains the aspect ratio of an image by default, Flex sizes the image to 150 x 150 pixels, the largest possible image that maintains the aspect ratio and conforms to the size constraints. The other 50 x 150 pixels remain empty. However, they are reserved by the `<mx:Image>` tag and unavailable to other controls and layout elements.

You can use a Resize effect to change the width and height of an image in response to a trigger. As part of configuring the Resize effect, you specify a new height and width for the image. Flex maintains the aspect ratio of the image by default, so it resizes the image as much as possible to conform to the new size, while maintaining the aspect ratio. For example, place your mouse pointer over the image in this example to enlarge it, and then move the mouse off the image to shrink it to its original size:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:Effect>
    <mx:Resize name="resizeBig" widthTo="150" duration="500"/>
    <mx:Resize name="resizeSmall" widthTo="120" duration="500"/>
  </mx:Effect>
  <mx:Image width="120" source="@Embed('../assets/icecreampint.jpg')"
    mouseOverEffect="resizeBig" mouseOutEffect="resizeSmall"/>
</mx:Application>
```

Note: You can find this example in the Effects and Behaviors section of the Flex Explorer sample application.

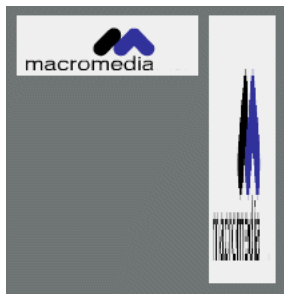
For more information on the Resize effect, see [Chapter 18, “Using Behaviors,” on page 453](#).

If you do not want to preserve the aspect ratio when you resize an image, you can set the `maintainAspectRatio` property to `false`. By default, `maintainAspectRatio` is set to `true` to enable the preservation of the aspect ratio.

The following example resizes your square image to the exact values of the height and width properties:

```
<mx:Image source="myImage.jpg" height="150" width="200"
  maintainAspectRatio="false" />
```

By choosing not to maintain the aspect ratio, you allow for the possibility of a distorted image. For example, the Macromedia logo is 136 x 45 pixels by default. In the following example, it is distorted beyond recognition because the aspect ratio is not maintained when the image is resized:



You can try this with your own logo or image using the following code:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:VBox id="vbox0" >
    <mx:Image source="@Embed('logo.gif')" />
    <mx:Image source="@Embed('logo.gif')" maintainAspectRatio="false"
      height="200" width="50" />
  </mx:VBox>
</mx:Application>
```

```

    </mx:VBox>
</mx:Application>

```

Positioning the image in a Canvas container

A Canvas container lets you specify the location of its children within the container. To specify the absolute position of an image, you use the `x` and `y` properties.

Note: In all other containers except the Canvas container, the container controls the positioning of its children and ignores the `x` and `y` properties. Use Canvas containers in conjunction with other containers if you need precision positioning.

The `x` and `y` properties specify the location of the upper left corner of the image in the Canvas container. In the following example, you set the position of the image at (10,10), 10 pixels down and 10 pixels to the right of the upper left corner of the Canvas container:

```

<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
    <mx:Canvas id="vbox0" >
        <mx:Image id="img0" source="myImage.jpg" x="10" y="10" />
    </mx:Canvas>
</mx:Application>

```

You can use a Canvas container for more complex positioning of multiple images. The following example creates a small checkerboard:

```

<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
    <mx:Canvas id="chboard" >
        <mx:Image source="BlackBox.jpg" width="10" height="10" x="0" y="0" />
        <mx:Image source="BlackBox.jpg" width="10" height="10" x="20" y="0" />
        <mx:Image source="BlackBox.jpg" width="10" height="10" x="40" y="0" />
        <mx:Image source="BlackBox.jpg" width="10" height="10" x="10" y="10" />
        <mx:Image source="BlackBox.jpg" width="10" height="10" x="30" y="10" />
        <mx:Image source="BlackBox.jpg" width="10" height="10" x="0" y="20" />
        <mx:Image source="BlackBox.jpg" width="10" height="10" x="20" y="20" />
        <mx:Image source="BlackBox.jpg" width="10" height="10" x="40" y="20" />
        <mx:Image source="BlackBox.jpg" width="10" height="10" x="10" y="30" />
        <mx:Image source="BlackBox.jpg" width="10" height="10" x="30" y="30" />
        <mx:Image source="BlackBox.jpg" width="10" height="10" x="0" y="40" />
        <mx:Image source="BlackBox.jpg" width="10" height="10" x="20" y="40" />
        <mx:Image source="BlackBox.jpg" width="10" height="10" x="40" y="40" />
    </mx:Canvas>
</mx:Application>

```

Flex renders the example as follows:



The Canvas container is sized to fit the images defined, and then positioned within the application window according to its default alignment and margin settings (vertically aligned to top, horizontally aligned to center, 24-pixel top margin, and 24-pixel bottom margin).

Setting visibility

The `visible` property lets you load an image but render it invisible. By default, the image is visible. To make an image invisible, set the `visible` property to `false`. The following example loads an image but does not make it visible:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:VBox id="vbox0" height="50" width="50">
    <mx:Image id="img0" source="myImage.jpg" visible="false"/>
  </mx:VBox>
</mx:Application>
```

The `VBox` container still allocates space for the image when it lays out its children. Therefore, if your application contained a `Button` control after the `<mx:Image>` tag, the button would appear in the same location as if the image was visible.

Often, you use the `visible` property to mark all images except one image invisible. For example, assume that you have an area of your application dedicated to showing one of three possible images based on some user action. Only one of the possible images would have its `visible` property set to `true`; all other images would be invisible by having their `visible` property set to `false`.

You can use `ActionScript` to set image properties. In the following example, you include a button that, when clicked, sets the `visible` property of the image to `true` to make it appear:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:Script>
    <![CDATA[
      function showImage(){
        image1.visible=true;
      }
    ]]>
  </mx:Script>

  <mx:VBox id="vbox0" width="600" height="600">
    <mx:Image id="image1" source="atom.jpg" visible="false" />
    <mx:Button id="myButton" label="Show" click="showImage()"/>
  </mx:VBox>
</mx:Application>
```

If you set the `visible` property, you can control when images are loaded; by allocating space but making images invisible when a page loads, you ensure that any performance hit occurs during the initialization stage when users expect it, rather than as they interact with the application and perform actions that require the image. Setting the `visible` property also prevents awkward and unpleasant resizing and layout reorganization within your application at seemingly random intervals.

Using the Loader control

Often in a product catalog, when a user clicks an item, the catalog displays an image of the item. One strategy for building a catalog is to load the catalog images into your application, but make them all invisible. When a user selects a product, you make that image visible.

However, this strategy requires that you insert an `<mx:Image>` tag for all the images in your catalog and load the images, even if they are invisible, at application startup. The resulting SWF file would be unnecessarily large, because it would have to contain all the images and its start-up time would be negatively affected by loading invisible images.

A better strategy is to dynamically load the images from your server, as necessary. In this way, your SWF file stays small, because it does not have to contain invisible images, and your start-up time improves.

As part of its implementation, the `ActionScript` class that defines the `<mx:Image>` tag is a subclass of the `Flex Loader` class. After creating an image, you can use the properties and methods of the `Loader` control with your image, including the `load()` method, which loads a SWF or JPEG file.

Note: The `load()` method of the `Loader` control only works with SWF and JPEG files; you cannot use it to load PNG, GIF, or SVG files.

The following example uses the `load()` method to replace the `atom.jpg` image with the `atomAfter.jpg` image when the user clicks a button:

```
<mx:Script>
    <![CDATA[
        function afterImage(){
            image1.load('atomAfter.jpg');
        }
    ]]>
</mx:Script>

<mx:VBox id="vbox0" width="600" height="600">
    <mx:Image id="image1" source="atom.jpg" />
    <mx:Button id="myButton" label="Show After" click="afterImage()"/>
</mx:VBox>
```

The container that holds the image does not adjust the layout of its children when you call the `load()` method. Therefore, you typically replace one image with another image of the same size. If the new image is significantly larger than the original, it can overlay other components in the container.

You can make the selection of the replacement image based on a user action in your application. For example, you might want to load an image based on a user selection in a list box or data grid.

In the next example, you use the index number of the selected item in a data grid to determine the image to load. In this example, images are named `1.jpg`, `2.jpg`, `3.jpg`, and so on, corresponding to items in the grid.

```
// Retrieve the image associated with the item selected in the grid.
function getImage() {
    var cartGrid = dgrid;
    var imageSource:String = 'images/' + cartGrid.getSelectedIndex() + '.jpg';
    image1.load(imageSource);
}
```

In this example, the images are stored in the `images` directory. The complete path to an image is the directory name, the index number, and the file suffix `.jpg`.

You register this function as the event handler for a change event in the data grid as follows:

```
<mx:DataGrid id="dgrid" height="200" width="350" change="getImage()"/>
```

When a user changes the currently selected item in the data grid, Flex calls the `getImage()` function to update the displayed image.

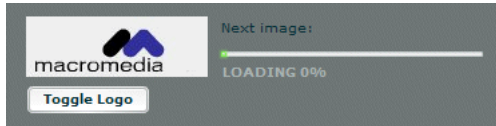
You could modify this example to use information about the selected item to determine the image to load, rather than using the selected item's index. For example, the grid could contain a list of objects, where each object has a property that contains the image name associated with it.

In addition to the `load()` method, you can also access the properties of the Loader control, including `contentPath` and `percentLoaded`. For a complete list of the Loader properties and methods, see *Flex ActionScript and MXML API Reference*. The `percentLoaded` property is particularly useful because it lets you to display a progress bar so users know that the application did not become unresponsive.

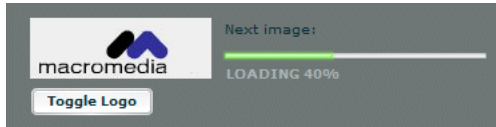
For example, the following simple application lets you to toggle between two different logos. It displays a progress bar to the right of the images.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:Script>
    <![CDATA[
      var first:String = "true";
      function newImage(){
        if (first=="true"){
          img1.load('logo2.jpg');
          first="false";
        }
        else {
          img1.load('logo.jpg');
          first="true";
        }
      }
    ]]>
  </mx:Script>
  <mx:HBox>
    <mx:VBox>
      <mx:Image id="img1" source="logo.jpg"/>
      <mx:Button id="button1" label="Toggle Logo" click="newImage()"/>
    </mx:VBox>
    <mx:VBox>
      <mx:Label text="Next image:"/>
      <mx:ProgressBar mode="manual" width="200"
        initialize="event.target.setProgress(img1.percentLoaded,100)"/>
    </mx:VBox>
  </mx:HBox>
</mx:Application>
```

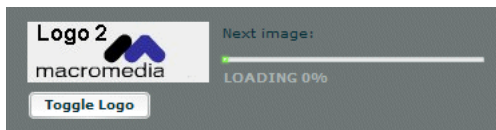
This example results in an application that looks like the following:



When you click the Toggle Logo button, the progress bar tracks the percentage of the new logo loaded, as the following figure shows:



After the image is fully loaded, the progress bar returns to its original state:



Click the Toggle Logo button a second time to reload the original image.

Referencing images in other MXML tags

Many Flex tags, such as `<mx:Button>` and `<mx:TabNavigator>`, take an `icon` property or other property that lets you specify an image file for the control to use. You must embed the images used in these MXML tags must be embedded; you cannot use runtime importing. You can use any supported graphic file in these tags, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:Button id="button1" icon="@Embed('myPic.svg')" />
  <mx:Button id="button2" icon="@Embed('myPic.jpg')" />
  <mx:Button id="button3" icon="@Embed('myPic.swf#symbol')"/>
</mx:Application>
```

The following table lists the properties that support image import:

| MXML tag | Property |
|--------------------------------|-------------------------------|
| <code><mx:Alert></code> | <code>icon</code> |
| <code><mx:Button></code> | <code>icon</code> |
| <code><mx:Link></code> | <code>icon</code> |
| <code><mx:Loader></code> | <code>brokenImage</code> |
| <code><mx:Tree></code> | <code>defaultLeafIcon</code> |
| <code><mx:Tree></code> | <code>folderOpenIcon</code> |
| <code><mx:Tree></code> | <code>folderClosedIcon</code> |

Importing images using Embed in ActionScript

You can embed images within ActionScript using the `[Embed]` compiler directive. This is functionally equivalent to the MXML `@Embed` syntax, and you can use it to import JPEG, GIF, PNG, SVG, and SWF files at compile time. You can also import image assets from SWC files.

This syntax allows you to assign an imported image to a variable for repeated later use. It also allows you to change the image at runtime by reassigning the variable. Generally, this method of image import provides more flexibility than other mechanisms.

The following example loads an image file and assigns it to the `image1` variable:

```
[Embed(source="bar.gif")]
var image1:String;
```

The embedded resource does not have to be a locally stored asset. You can specify network URIs as the source, as the following example shows:

```
[Embed(source="http://getpic.com/getsvg")]
var pic:String;
```

If the source files of the external resources change after the Flex application is compiled, Flex will not recompile the application. Flex does not examine the source files on each request to determine if they are up to date.

Embedding symbols

You can embed symbols from inside SWF files using the `symbol` property of `[Embed]`. The following example extracts the symbol `x` from the `whee.swf` file and includes it in the Flex application:

```
[Embed(source="whee.swf", symbol="x")]
var symbolInsideWhee:String;
```

If the symbol has dependencies, Flex imports them as well; otherwise, Flex imports only the specified symbol from the SWF or SWC source file.

You can also use a number sign (`#`) to delimit a resource from a symbol name when you import assets from a SWC or SWF file, as the following example shows:

```
[Embed("myMovie.swf#image3")]
var myImage3:String;
```

If you embed an entire SWF file, the embedded SWF file cannot contain any ActionScript 2.0 class definitions or Macromedia components. If it does, Flex does not embed the SWF file. If you embed only a symbol from the SWF file, the SWF file can contain ActionScript 2.0 classes and components.

Specifying MIME types

You can optionally specify a MIME type for the imported asset with the `mimeType` property. If you do not specify a `mimeType` property, Flex makes a best guess about the type of file imported based on the file extension. The `mimeType` property overrides the default guess of the resource type.

Flex currently supports the following image MIME types:

- `image/gif`
- `image/jpeg`
- `image/png`
- `application/x-shockwave-flash`
- `application/x-macromedia-swc`
- `image/svg`
- `image/svg+xml`

The following example points to an ambiguously typed `source` property and specifies a MIME type so that Flex uses the proper transcoder to import the resource:

```
[Embed(source="http://getpic.com/getsvg", mimeType="image/svg")]
var pic:String;
```

Embed example

The following example shows you how to use `[Embed]` within your applications. It changes the cursor to a *wait* cursor during the loading of a large image file. When the load completes, the application removes the wait cursor and returns the cursor to the system cursor.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:Script>
    <![CDATA[
      import mx.managers.CursorManager;
      // Define a variable to hold the cursor ID.
      var cursorID : Number = 0;
      // Embed the cursor symbol.
      [Embed(source="wait.jpg")]
      var waitCursorSymbol:String;

      // Define event handler to display the wait cursor and to load the image.
      function initImage(event) {
        cursorID = CursorManager.setCursor(waitCursorSymbol);
        image1.load("DSC00034.JPG");
      }

      // Define an event handler to remove the wait cursor.
      function hideCursor(event){
        CursorManager.removeCursor(cursorID);
      }
    ]]>
  </mx:Script>
```

```
<mx:VBox>
  <!-- Loader control to load the image. -->
  <mx:Loader id="image1" complete="hideCursor(event)" />
  <!-- Button triggers the load. -->
  <mx:Button id="myButton" label="Show" click="initImage(event)"/>
</mx:VBox>
</mx:Application>
```

For more information about working with cursors, see [Chapter 21, “Using the Cursor Manager”](#).

CHAPTER 10

Importing Media

Macromedia Flex supports the `MediaDisplay`, `MediaController`, and `MediaPlayback` controls to incorporate streaming media into Flex applications. Flex supports the Macromedia Flash Video File (FLV) and MP3 file formats with these controls. This chapter describes how to use the media controls in your application.

Contents

| | |
|--|-----|
| Using Media in Flex. | 297 |
| Importing MP3 files | 298 |
| About the <code>MediaDisplay</code> control | 299 |
| About the <code>MediaController</code> control | 300 |
| About the <code>MediaPlayback</code> control | 302 |
| Sizing a media component | 302 |
| Adding a cue point. | 302 |
| Syntax for the media controls. | 303 |

Using Media in Flex

Media, such as movie and audio clips, are used more and more to provide information to web users. As a result, you need to provide users with a way to stream the media, and then control it. The following examples are usage scenarios for media controls:

- Showing a video message from the CEO of your company
- Streaming movies or movie previews
- Streaming songs or song snippets
- Providing learning material in the form of media

The Flex streaming media controls make it easy to incorporate streaming media into Flash presentations. Flex supports the Flash Video File (FLV) and MP3 file formats with these controls.

You can use the following media controls:

MediaDisplay control Enables streaming media in your application without a supporting user interface. You can use this control with video and audio data. When you use the MediaDisplay control by itself your application provides no mechanism for its users to control the media files.

MediaController control Complements the MediaDisplay control by providing a user interface that controls media playback using standard controls, such as play and pause. You only use this control with the MediaDisplay control, not with the MediaPlayer control. The MediaController control features a drawer, which exposes the playback controls when the user positions the mouse pointer over them.

MediaPlayer control A combination of the MediaDisplay and MediaController controls, the MediaPlayer control provides methods to stream your media content and an interface that lets users control playback.

Note: None of the media controls support scan forward and scan backward functionality. However, you can achieve this functionality by moving the playback sliders. In addition, the media controls do not support accessibility or styles.

Importing MP3 files

In addition to these streaming mechanisms, Flex lets you import MP3 files using the [Embed] syntax. You can then play these files using a Sound object. Use the following syntax to import an MP3 file into your application:

```
[Embed('music.mp3')]
var myMusic:String;
```

If your MP3 file does not have a clear file extension, you must set the `mimeType` property of [Embed] to `audio/mpeg`, as follows:

```
[Embed(source="http://musicdownloads.com/music", mimeType="audio/mpeg")]
var myMusic:String;
```

Importing versus streaming MP3 files

If you use this syntax, you are literally importing your MP3 files into your application. This has some advantages over streaming, such as instantaneous access to the files without any load times, but also some disadvantages, such as an increase in your application size and the initial application load time.

You can import either local files or files downloaded from a web server. However, if the source files of the external resources change after the Flex application is compiled, Flex will not recompile the application. To ensure that your application always uses the most current MP3 files, you must use the streaming mechanisms rather than importing the files into your application. Flex cannot stream local files; you must import them into your applications using [Embed].

Embed example

The following example application imports an MP3 file and adds two buttons to start and stop playing the MP3 file. It shows the use of the Sound object in ActionScript, as well as the use of the Embed metadata keyword to import the MP3 file.

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:Script>
    <![CDATA[
      //Create global variable.
      var snd:Sound;

      //Use Embed metadata keyword to point to resource.
      [Embed('bluechristmas.mp3')]
      //Declare global variable for the symbol.
      var soundSymbol:String;

      function startSound() {
        //Declare instance of Sound.
        snd = new Sound;
        //Attach sound symbol.
        snd.attachSound(soundSymbol);
        snd.start();
      }

      function stopSound() {
        snd.stop();
      }

    ]]>
  </mx:Script>

  <mx:VBox>
    <mx:Button label="Start" id="b1" click="startSound();" />
    <mx:Button label="Stop" id="b2" click="stopSound();" />
  </mx:VBox>
</mx:Application>
```

About the MediaDisplay control

Flex creates a MediaDisplay control with no visible user interface. It is simply a control to hold and play media.

Note: The user cannot see anything unless some video media is playing.

The appearance of any video media playing in a MediaDisplay control is affected by the following properties:

- aspectRatio
- autoSize
- height
- width
- volume

When you set `aspectRatio` to `true` (the default), the control adjusts the size of the playing media after the control size has been set to maintain the aspect ratio of the media.

If you omit both `width` and `height` properties for the control, Flex makes it the size of the video. If one is specified but not the other, the unspecified one is taken from the size of the video.

The `MediaDisplay` control also supports the `volume` property. This property takes an integer value from 0 to 100, with 0 being mute and 100 being the maximum volume. The default setting is 75.

The following example creates a `MediaDisplay` control:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >\
  <mx:VBox>
    <mx:MediaDisplay contentPath="http://myhost.com/media/MyVideo.flv"
      height="400" width="400" />\
  </mx:VBox>
</mx:Application>
```

About the `MediaController` control

The interface for the `MediaController` control depends on its `controllerPolicy` and `backgroundStyle` properties. The `controllerPolicy` property determines if the media control set is always expanded, always collapsed, or only expanded when the user hovers the mouse pointer over the control portion of the control.

When collapsed, the controller draws a modified progress bar. It shows the progress of the bytes being loaded at the bottom of the bar, and the progress of the playhead just above it.

The following figure shows the `MediaController` control in its expanded state:



The expanded state draws an enhanced version of the control, which contains the following items:

- Text labels on the left that indicate the playback state (streaming or paused)
- Text labels on the right that indicate playhead location, in seconds
- Playhead location indicator that users can drag to navigate within the media

The `MediaController` control also has the following items:

- A Play/Pause state button
- A Go to Beginning button which navigates to the beginning of the media
- A Go to End button which navigate to the end of the media
- A volume control that consists of a slider, a mute button, and a maximum volume button

Note: When Flex calculates the size of the control, it includes the size of the expanded control area so that it does not overlap other components when it expands.

Both the collapsed and expanded states of the `MediaController` control use the `backgroundStyle` property. This property determines whether the control draws a chrome background (the default) or lets the movie background appear from behind the controls.

The `MediaController` control has an orientation setting, `horizontal`, which you can use to draw the control with a horizontal orientation (the default) or a vertical one. With a horizontal orientation, the play bar tracks playing media from left to right. With a vertical orientation, the play bar tracks the media from bottom to top.

You use the `associatedDisplay` and `associatedController` properties in MXML, and the `associateDisplay()` and `associateController()` methods in `ActionScript`, to associate the `MediaDisplay` and `MediaController` controls with each other.

When you associate a `MediaController` control with a `MediaDisplay` control, the `MediaController` control updates its controls based on events broadcast from the `MediaDisplay` control, and lets the `MediaDisplay` control react to the settings made by the user from the `MediaController` control.

The following example uses the MXML `associatedDisplay` property to associate the two controls:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:VBox>
    <mx:MediaController id="myMC" associatedDisplay="myMD"
      controllerPolicy="on" backgroundStyle="default" />
    <mx:MediaDisplay id="myMD"
      contentPath="http://myhost.com/media/MyVideo.flv"
      height="400" width="400" autoPlay="false" />
  </mx:VBox>
</mx:Application>
```

The order in which you define the controls defines the order in which they appear in your application. In this example, the `MediaController` control appears above the `MediaDisplay` control. Reversing the order in which you define them in the MXML file reverses the order in which they appear in your application.

The following example creates a `MediaDisplay` control, and associates it with a `MediaController` control. You call either the `associateDisplay()` or the `associateController()` method, not both.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:VBox>
    <mx:MediaDisplay id="myMD"
      contentPath="http://myhost.com/media/MyVideo.flv"
      height="400" width="400" autoPlay="false"
      initialize="myMD.associateController(myMC)" />
    <mx:MediaController id="myMC" controllerPolicy="on" />
  </mx:VBox>
</mx:Application>
```

In this example, you create a `MediaController` control and configure it so that the controls are always visible.

About the MediaPlayer control

The MediaPlayer control is a combination of the MediaController and MediaDisplay controls. The MediaPlayer control uses the `controlPlacement` property to determine the layout of the controls. The possible control placements include `top`, `bottom`, `left`, and `right`, indicating where the controls are drawn in relation to the display. For example, a value of `right` gives a control a vertical orientation and positions it on the right of the display.

The following example creates a MediaPlayer control:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

    <mx:VBox>

        <mx:MediaPlayer contentPath="http://myhost.com/media/MyVideo.flv"
            height="400" width="400" controllerPolicy="on" controlPlacement="left"
            autoPlay="false" />

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

In this example, you display an FLV file in the control and start the control in the paused state. The control bar appears on the left side of the control, and it is always visible.

Sizing a media component

By default, the MediaDisplay and MediaPlayer controls size themselves to the size of the media. If you specify the `width` or `height` property to the control, and either is smaller than the media's dimensions, the component does not size itself to the size of the media. Instead, Flex sizes the media to fit within the component.

If Flex resizes the media, the `aspectRatio` and `autoSize` properties of the control determine how to perform the resizing. If `aspectRatio` is `true`, which is the default, the media retains its aspect ratio when resizing.

If you set the `autoSize` property to `true`, Flex displays the media at its preferred size, unless the control's playback area is smaller than the preferred size. In that case, Flex shrinks the media to fit inside the control. The default value is `true`.

Adding a cue point

You can use cue points to trigger events when the playback of your media reaches a specified location. To set cue points, you pass an array to the `cuePoints` property of the MediaDisplay or MediaPlayer controls. Each element of the array contains two fields. The `name` field contains an arbitrary name of the cue point. The `time` field contains the playhead location, in seconds, of the MediaPlayer or MediaDisplay control with which the cue point is associated.

When the playhead of the MediaPlayer or MediaDisplay control reaches a cue point, it broadcasts a `cuePoint` event, as the following example shows:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
```

```

<mx:Script>
  <![CDATA[
    function cpHandler(event){
      cp.text="go to cuepoint: " + evt.cuePointName + " " +
        evt.cuePointTime;
    ]]>
</mx:Script>
<mx:VBox>
  <mx:MediaPlayback contentPath="//myhost.com/media/MyVideo.flv/MyMP3.MP3"
    cuePoint="cpHandler()">
    <mx:cuePoints>
      <mx:Array>
        <mx:Object name="first" time="10" />
        <mx:Object name="second" time="20" />
      </mx:Array>
    </mx:cuePoints>
  </mx:MediaPlayback>
  <mx:TextArea id="cp" />
</mx:VBox>
</mx:Application>

```

In this example, the event handler writes a text string to the `TextArea` control when the control reaches a cue point. The string contains the name and time of the cue point.

Syntax for the media controls

You use the `<mx:MediaDisplay>`, `<mx:MediaController>`, and `<mx:MediaPlayback>` tags to define media controls. These controls inherit all the properties of the `UIComponent` class. For a list of these properties, see [Chapter 1, “Using Flex Components,” on page 15](#).

The following table lists the optional properties that are defined by these controls:

| Property | Control | Type | Use | Description |
|-----------------------------------|---|---------|----------|---|
| <code>activePlayControl</code> | <code>MediaController</code> | Boolean | Property | Specifies the play state of the associated <code>MediaDisplay</code> control when loaded at runtime. The default value is <code>true</code> , to play the media file when loading. Set this property to the same value as the <code>autoPlay</code> property of the associated <code>MediaDisplay</code> control. |
| <code>aspectRatio</code> | <code>MediaDisplay</code> <code>MediaPlayback</code> | Boolean | Property | Specifies whether the control instance maintains its video aspect ratio, <code>true</code> ; otherwise <code>false</code> . The default value is <code>true</code> . |
| <code>associatedController</code> | <code>MediaDisplay</code> | String | Property | Specifies a <code>MediaController</code> control associated with the <code>MediaDisplay</code> control. |
| <code>associatedDisplay</code> | <code>MediaController</code> | String | Property | Specifies a <code>MediaDisplay</code> control associated with the <code>MediaController</code> control. |

| Property | Control | Type | Use | Description |
|-------------------------------|----------------------------------|---------|----------|--|
| <code>autoPlay</code> | MediaDisplay MediaPlayback | Boolean | Property | Specifies whether the control immediately starts to buffer and play, <code>true</code> ; otherwise <code>false</code> . The default value is <code>true</code> . For a MediaDisplay control, set this property to <code>true</code> when the <code>activePlayControl</code> property of any associated MediaController control is set to <code>play</code> . Set this property to <code>false</code> when the <code>activePlayControl</code> property of any associated MediaController control is set to <code>pause</code> . |
| <code>autoSize</code> | MediaDisplay MediaPlayback | Boolean | Property | Specifies whether Flex displays the media at its preferred size, <code>true</code> , unless the control's playback area is smaller than the preferred size. In that case, Flex shrinks the media to fit inside the control. The default value is <code>true</code> . |
| <code>backgroundStyle</code> | MediaController | String | Property | Specifies whether the control draws its chrome background, <code>default</code> ; otherwise, <code>none</code> . The default value is <code>default</code> . |
| <code>bytesLoaded</code> | MediaDisplay MediaPlayback | Number | Property | Read-only property that contains the total number of bytes loaded that are available for playing. |
| <code>bytesTotal</code> | MediaDisplay MediaPlayback | Number | Property | Read-only property that contains the total number of bytes to load into the control. |
| <code>contentPath</code> | MediaDisplay MediaPlayback | String | Property | Specifies the absolute or relative URL of the media to be streamed and played. |
| <code>controllerPolicy</code> | MediaController MediaPlayback | String | Property | Specifies whether the control is hidden and only appears when the user moves the mouse pointer over the controller's collapsed state. The following are the possible values for this property: <ul style="list-style-type: none"> <code>on</code> The control is always expanded. <code>off</code> The control is always collapsed. <code>auto</code> (default) The control remains in the collapsed state until the user moves the mouse pointer over the hit area. The hit area matches the area in which the collapsed control is drawn. The control remains expanded until the user moves the mouse pointer off of the hit area. |
| <code>controlPlacement</code> | MediaPlayback | String | Property | Specifies where the controller portion of the MediaPlayback control is positioned in relation to its display. The possible values are <code>top</code> , <code>bottom</code> , <code>left</code> , and <code>right</code> . The default value is <code>bottom</code> . |

| Property | Control | Type | Use | Description |
|------------|-------------------------------|---------|----------|--|
| cuePoints | MediaDisplay MediaPlayback | Array | Property | <p>Specifies an array of cue point objects that is assigned to a given control instance.</p> <p>The name of the cue point is arbitrary and can be used as references to determine which functions are fired. For example:</p> <pre>function cpHandler(event){ if(event.cuePoint.name == "fireSaleAnnouncement"){ //pop up sale announcement } else if(event.cuePoint.time == 30){ //pop up similar movies list } }</pre> <p>The <code>time</code> property specifies the playhead location of the <code>MediaPlayback</code> or <code>MediaDisplay</code> control with which the cue point is associated.</p> <p>The <code>fps</code> value can affect how cue points work. For more information, see the entry for <code>fps</code>.</p> |
| fps | MediaDisplay MediaPlayback | Number | Property | <p>Specifies the frames-per-second of the video. Flex uses this value to calculate milliseconds based on frame number; it does not actually affect the display of the video.</p> <p>The default value is 30.</p> <p>The <code>fps</code> value does affect how cue points with fractional values are interpreted. If the fractional portion of the cue point value is less than the <code>fps</code> value, it is interpreted as the number of frames past the specified seconds (the integer portion of the value). If the fraction portion of the cue point value is greater than the <code>fps</code> value, it is interpreted as microseconds. For example, if <code>fps=12</code> and <code>cuePoint=11.11</code>, the cue point is at 11 frames after the 11-second mark. If <code>fps=10</code> and <code>cuePoint=11.11</code>, the cue point is at 11.000011 seconds.</p> |
| horizontal | MediaController | Boolean | Property | <p>Specifies the orientation of the control instance as horizontal, true, or vertical, false. The default value is true.</p> |

| Property | Control | Type | Use | Description |
|-------------------------------------|--------------------------------|---------|----------|--|
| <code>mostRecentCuePoint</code> | MediaDisplay MediaPlayback | Object | Property | Read-only property that contains a reference to the <code>cuePoint</code> object of the most recent cue point. To retrieve the name and time of the cue point, use: <code>mostRecentCuePoint.name</code> <code>mostRecentCuePoint.time</code> |
| <code>mostRecentCuePointName</code> | MediaDisplay MediaPlayback | String | Property | Read-only property that contains the name of the most recent cue point. |
| <code>mostRecentCuePointTime</code> | MediaDisplay MediaPlayback | Number | Property | Read-only property that contains the time, in seconds, within the playback media of the most recent cue point. |
| <code>mediaType</code> | MediaDisplay MediaPlayback | String | Property | Read-only property that contains the type of media being played: Flash video (FLV) or MP3 file (MP3). |
| <code>playheadTime</code> | MediaDisplay MediaPlayback | Number | Property | Property that contains the current position of the playhead, in seconds. |
| <code>playing</code> | MediaDisplay, MediaPlayback | Boolean | Property | Read-only property that contains <code>true</code> if the control is playing media. |
| <code>totalTime</code> | MediaDisplay MediaPlayback | Number | Property | Read-only property that contains the total length of the media, in seconds, for an MP3 or FLV 1.1 or higher file, or <code>undefined</code> for an FLV 1.0 file. |
| <code>videoHeight</code> | MediaDisplay | Number | Property | Read-only property that contains the true height of the video display. |
| <code>videoWidth</code> | MediaDisplay | Number | Property | Read-only property that contains the true width of the video display. |
| <code>volume</code> | MediaDisplay MediaPlayback | Number | Property | Specifies an integer from 0 to 100 that represents the volume level. The default value is 75. |

PART II

Improving User Experience

This part describes how to improve the user experience by adding additional functionality to your application.

The following chapters are included:

| | |
|--|-----|
| Chapter 11: Building an Application with Multiple MXML Files | 309 |
| Chapter 12: Working with ActionScript in Flex | 319 |
| Chapter 13: Using Events | 335 |
| Chapter 14: Creating ActionScript Components | 359 |
| Chapter 15: Customizing Data Provider Controls | 379 |
| Chapter 16: Using Styles and Fonts | 395 |
| Chapter 18: Using Behaviors | 453 |
| Chapter 19: Creating Charts in Flex | 475 |
| Chapter 20: Using ToolTips. | 539 |
| Chapter 21: Using the Cursor Manager | 547 |
| Chapter 22: Using the Drag and Drop Manager | 553 |
| Chapter 23: Using the History Manager | 569 |
| Chapter 24: Improving Startup Performance | 577 |
| Chapter 25: Using Runtime Shared Libraries. | 595 |
| Chapter 26: Printing from SWF Files. | 607 |
| Chapter 27: Creating Accessible Applications | 617 |

CHAPTER 11

Building an Application with Multiple MXML Files

This chapter describes how to use MXML files as custom tags in other MXML files. MXML components provide an easy way to extend an existing Macromedia Flex component and encapsulate the appearance and behavior of a component in a custom MXML tag.

Contents

| | |
|------------------------------------|-----|
| About MXML components | 309 |
| Creating MXML components | 311 |
| Passing component references | 316 |
| Using interfaces | 317 |

About MXML components

MXML components are MXML files that you use as custom tags in other MXML files. They encapsulate and extend the functionality of existing Flex components.

Using MXML components promotes code reuse, simplifies the process of building a complex application, and makes it easier for more than one developer to contribute to a project.

Using MXML components

An application that uses MXML components consists of an MXML application file with an `<mx:Application>` root tag that references one or more components defined in separate MXML files. Each MXML component extends an existing Flex component or another MXML component.

You create an MXML component in an MXML file with the component tag name. For example, a file named `MyForm.mxml` defines a component named `MyForm`. Flex uses the spelling and capitalization of the filename to generate a new class that represents the component.

The root tag of an MXML component is the parent component tag. For example, the following MXML component extends the standard Flex `ComboBox` control. The root tag specifies the `http://www.macromedia.com/2003/mxml` namespace.

```
<?xml version="1.0"?>
<!-- MyComboBox.mxml -->
```

```

<mx:ComboBox xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:dataProvider>
    <mx:Array>
      <mx:String>AK</mx:String>
      <mx:String>AL</mx:String>
    </mx:Array>
  </mx:dataProvider>
</mx:ComboBox>

```

The application in the following example uses the MyComboBox component:

```

<?xml version="1.0"?>
<!-- MyApplication.mxml -->

<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
  xmlns:local="*">

  <local:MyComboBox/>

</mx:Application>

```

Referencing MXML components

Depending on where an MXML component is located, you refer to it in one of the following ways:

- If you have components in the same directory as the application file or in an ActionScript classpath directory (not a subdirectory) defined in the flex-config.xml file, you can refer to the components as the following example shows. In this example, the local namespace (*) is mapped to the prefix local.

```

<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
  xmlns:local="*" >

  <local:MyButton />

</mx:Application>

```

If the same file exists in an ActionScript classpath directory and the application directory, Flex uses the file in the application directory.

- If you have components in a subdirectory of the directory that contains the application file or in a subdirectory of the ActionScript classpath directory defined in the flex-config.xml file, you can refer to the components as the following example shows. In this example, the foo.bar namespace is mapped to the comp prefix. The MyButton.mxml file is located in the foo/bar directory.

```

<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
  xmlns:comp="foo.bar.*" >

  <comp:MyButton />

</mx:Application>

```

If the same file exists under an ActionScript classpath subdirectory and the application subdirectory, Flex uses the one under the application subdirectory.

The `WEB-INF/flex/user-classes` directory is the default ActionScript classpath directory. If you plan to share components between applications, you should place them in the ActionScript classpath directory. You configure the ActionScript classpath in the `<actionscript-classpath>` tag in the `flex-config.xml` file. The following example shows the default `<actionscript-classpath>` tag:

```
<compiler>
  <actionscript-classpath>
    <path-element>/WEB-INF/flex/user_classes</path-element>
  </actionscript-classpath>
</compiler>
```

Note: File lookup is case-sensitive on all file systems. On case-insensitive file systems, such as the Macintosh and Windows file systems, the Flex compiler generates a case-mismatch error when you use a component with the incorrect case. On case-sensitive file systems, such as the UNIX file system, the Flex compiler generates a component-not-found error when you use a component with the incorrect case.

Creating MXML components

You can create the following types of MXML components:

- Controls
- Containers
- Applications

This section describes how to create and use each type of component and how to add properties and methods to a custom MXML component.

Creating and using a control

To change the appearance or behavior of a standard Flex control, you can extend it by using its tag as the root tag of an MXML component file.

For example, the following custom `ComboBox` control presets a list of states in `<mx:String>` tags:

```
<?xml version="1.0"?>
<!-- StateComboBox.mxml -->

<mx:ComboBox xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:dataProvider>
    <mx:Array>
      <mx:String>AK</mx:String>
      <mx:String>AL</mx:String>
    </mx:Array>
  </mx:dataProvider>
</mx:ComboBox>
```

The following MXML application file references the `StateComboBox` control in the

`<StateComboBox>` tag:

```
<?xml version="1.0"?>
<!-- myapp.mxml -->
```

```

<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
  xmlns:local="*">

  <local:StateComboBox id="state"/>

</mx:Application>

```

Creating and using a container

To change the appearance or behavior of a standard Flex container, you can extend it by using its tag as the root tag of an MXML file. Containers typically contain child controls and other containers.

For example, the following component contains an address form. One of the `<mx:FormItem>` tags contains a `<local:StateComboBox>` tag:

```

<?xml version="1.0"?>
<!-- AddressForm.mxml -->

<mx:Form xmlns:mx="http://www.macromedia.com/2003/mxml" xmlns:local="*">

  <mx:FormItem label="Name">
    <mx:TextInput id="name" />
  </mx:FormItem>

  <mx:FormItem label="Street">
    <mx:TextInput id="street" />
  </mx:FormItem>

  <mx:FormItem label="City" >
    <mx:TextInput id="city" />
  </mx:FormItem>

  <mx:FormItem label="State" >
    <local:StateComboBox id="state"/>
  </mx:FormItem>

</mx:Form>

```

The following application file references the AddressForm component in the `<AddressForm>` tag:

```

<?xml version="1.0"?>
<!-- myapp.mxml -->

<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" xmlns:local="*"
  height="300">

  <local:AddressForm/>

</mx:Application>

```

Note: If you use child tags in an MXML component file, you cannot add child tags when you use the component as a custom tag in another MXML file. If you do not use child tags in an MXML component file, you can add child tags when you use the component as a custom tag.

Creating and using an application component

When you reference an MXML application file as an MXML component at the root of another MXML file, the MXML component inherits all of the styles and children of the original file.

For example, the following MXML application file declares a WebService object called `weatherService` in an `<mx:WebService>` tag:

```
<?xml version="1.0"?>
<!-- WeatherBase.mxml -->

<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
  <mx:WebService id="weatherService"
    wsdl="http://weather.unisysfsp.com/PDCWebService/
    WeatherServices.asmx?WSDL">
  </mx:WebService>
  ...
</mx:Application>
```

The following MXML file uses the WeatherBase component as its root tag, so it inherits the WebService declared in the WeatherBase.mxml file:

```
<?xml version="1.0"?>
<!-- weather1.mxml -->

<local:WeatherBase xmlns:local="*" xmlns:mx="http://www.macromedia.com/2003/
mxml">
  <mx:TextArea id="zipcode"/>
  <mx:Button id="go"/>
  <mx:Script>
    <![CDATA[
      go.click=function() {
        weatherService.GetWeather(zipcode.text);
        // weatherService is defined in the parent class
      }
    ]]>
  </mx:Script>
</local:WeatherBase>
```

Adding custom properties and methods to a component

You can define properties of MXML components in MXML tags or as `ActionScript` variables. You can define a method of an MXML component as an `ActionScript` function.

Defining properties in MXML tags

In MXML, you can use the `<mx:String>`, `<mx:Number>`, and `<mx:Boolean>` tags to define properties that take String, Number, or Boolean values, respectively. When using one of these tags, you must specify an `id`, which becomes the property name.

Optionally, you can specify an initial value in the body of the tag, or you can use the `source` property to specify the contents of an external URL or file as the initial property value. If you use the `source` property, the body of the tag must be empty. The initial value can be static data or a binding expression.

The following examples show initial properties set as static data and binding expressions; values are set in the tag bodies and in the source properties:

```
<!-- String property examples: -->
<mx:String id="myStringProperty">Welcome, {CustomerName}.</mx:String>

<mx:String id="myStringProperty1" source="http://www.somesite.com/file"/>

<!-- Number property examples: -->
<mx:Number id="myNumberProperty">15</mx:Number>

<mx:Number id="minutes">{numHours * 60}</mx:Number>

<!-- Boolean property examples: -->
<mx:Boolean id="myBooleanProperty">true</mx:Boolean>

<mx:Boolean id="passwordStatus">{passwordExpired}</mx:Boolean>
```

The MXML component in the following example contains a property defined in an `<mx:String>` tag:

```
<?xml version="1.0"?>
<!-- StateComboBox.mxml -->

<mx:ComboBox xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:String id="myStringProperty"/>
  <mx:dataProvider>
    <mx:Array>
      <mx:String>AK</mx:String>
      <mx:String>AL</mx:String>
    </mx:Array>
  </mx:dataProvider>
  ...
</mx:ComboBox>
```

The following MXML application file references the `StateComboBox` control in the `<local:StateComboBox>` tag:

```
<?xml version="1.0"?>
<!-- myapp.mxml -->

<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
  xmlns:local="*">

  <local:StateComboBox id="state" myStringProperty="Hello World" />

</mx:Application>
```

Defining properties and methods in ActionScript

You can define properties and methods of an MXML component in an `<mx:Script>` tag. The `<mx:Script>` tag must be an immediate child of the root tag of the MXML file. A public ActionScript function in an `<mx:Script>` tag becomes a method of the component. A public variable declaration or a set function in an `<mx:Script>` tag becomes a property of the component. You should precede the variable or set function with the `[Inspectable]` metadata tag if you plan to use the component in an authoring tool such as the Flex Builder tool. For more information about defining properties in ActionScript, see [Chapter 12, “Working with ActionScript in Flex,”](#) on page 319.

In the following example, the `MyComboBox` component contains a `MyNumberProperty` property, a `MyStringProperty` property, and a `getStatus()` method, in addition to the standard `ComboBox` methods and properties:

```
<?xml version="1.0"?>
<!-- MyComboBox.mxml -->

<mx:ComboBox xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:Script>
    <![CDATA[
      // This variable is a property of the component:
      public var myNumberProperty:Number;

      // This variable is a property of the component:
      public var myStringProperty:String;

      // This function is a method of the component.
      public function getStatus() {
        return "Selection was changed \nNumber is " + myNumberProperty;
      }
    ]]>
  </mx:Script>
  <mx:dataProvider>
    <mx:Array>
      <mx:String>AK</mx:String>
      <mx:String>AL</mx:String>
    </mx:Array>
  </mx:dataProvider>
</mx:ComboBox>
```

You can call a component's custom methods and access its properties just as you would any instance method or component property, as the following application shows:

```
<?xml version="1.0"?>
<!-- MyApplication.mxml -->
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
  xmlns:local="*">

  <local:MyComboBox id="mcb" myNumberProperty="4" change="ta1.text =
    mcb.getStatus();" />

  <mx:TextArea id="ta1" width="300" height="150" />

</mx:Application>
```

Declaring component metadata

In an ActionScript class, you can declare metadata tags inside brackets. The code in the following example creates a custom event:

```
[Event("MyEvent")]
```

In an MXML component, you can declare metadata tags in an `<mx:Metadata>` tag, as the following example shows:

```
<mx:Metadata>
    [Event("itemSelected")]
    [Event("checkOut")]
</mx:Metadata>
```

Passing component references

This section describes how to reference MXML components in other objects, and how to reference application objects in an MXML component.

Referencing an MXML component in other application objects

Just like the methods and properties of a standard component, you can access all of the methods and properties of an MXML component in an `<mx:Script>` block. Statements that call methods or set properties must be contained in the body of a function.

Properties have simple or complex types, which govern the syntax used to set them in MXML. If you set a property to a value in the MXML file that defines a component, you can override that value in an instance of that component used in another file.

Referencing application objects in an MXML component

To pass object references to an MXML component from other components in an application, you can create a property in the MXML component to represent the Application object. You can reference it by typing a property that is the same as the name of the root MXML file with the `<mx:Application>` tag.

In the following example, the MXML component contains a property called `app` that is of type `Apple`. `Apple.xml` is the name of the root application file. Creating a property of type `Apple` provides strong typing benefits and ensures that binding works correctly. The `text` property of the `mytext` `TextInput` control is set to the `text` property value of the `text1` `TextInput` control that is defined in the application file.

```
<?xml version="1.0"?>
<!-- MyComponent.xml -->

<mx:VBox label="User interface" xmlns:mx="http://www.macromedia.com/2003/
mxml">
    <mx:Script>
        <![CDATA[
            var app:Apple;
        ]]>
    </mx:Script>
```

```

    <mx:Button id="mybutton1" />
    <mx:TextInput id="mytext" text="{app.text1.text}"/>
</mx:VBox>

```

In the application file, you can bind the Application object to the MXML component's app property. In the following example, the MXML component has access to all of the Application object's children, including the text property value of the text1 TextInput control:

```

<?xml version="1.0"?>
<!-- Apple.mxml -->
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    xmlns:local="*">
    ...
    <local:MyComponent id="view1" app="{this}" />

    <mx:TextInput id="text1" text="Hello"/>
    ...
</mx:Application>

```

Using interfaces

Interfaces are a type of class that you design to act as an outline for your components. When you write an interface, you provide only the names of public methods rather than any implementation. If, for example, you define two methods in an interface and then implement that interface, the implementing class must provide implementations of those two methods.

Interfaces in ActionScript can only declare methods; they cannot specify constants. The benefits of interfaces are that you can define a contract that all classes implementing that interface must follow. In addition, if your class implements an interface, instances of that class can also be cast to that interface.

Custom MXML components can implement interfaces just as other ActionScript classes can. To do this, you use the `implements` attribute. All MXML tags support this attribute.

The following code is an example of a simple interface that declares several new methods:

```

// The following is in a file named SuperBox.as.
interface SuperBox {
    function selectSuperItem():String;
    function removeSuperItem():Boolean;
    function addSuperItem():Boolean;
}

```

A class that implements the SuperBox interface uses the `implements` attribute to point to its interface and must provide an implementation of the new methods. The following example of a custom ComboBox component implements the SuperBox interface:

```

<?xml version="1.0"?>
<!-- MyComboBox.mxml -->

<mx:ComboBox xmlns:mx="http://www.macromedia.com/2003/mxml"
    implements="SuperBox">
    <mx:Script>
        function selectSuperItem():String {
            return "Super Item was selected";
        }
    </mx:Script>

```

```

    }
    function removeSuperItem():Boolean {
        return true;
    }
    function addSuperItem():Boolean {
        return true;
    }
</mx:Script>
<mx:dataProvider>
    <mx:Array>
        <mx:String>AK</mx:String>
        <mx:String>AL</mx:String>
    </mx:Array>
</mx:dataProvider>
</mx:ComboBox>

```

You can implement multiple interfaces by separating them with commas as the following example shows:

```

<mx:ComboBox xmlns:mx="http://www.macromedia.com/2003/mxml"
    implements="SuperBox, SuperBorder, SuperData">

```

All methods declared in an interface are considered public. If you define an interface and then implement that interface but do not implement all of its methods, the MXML compiler throws an error.

Methods implemented in the custom component must have the same return type as their corresponding methods in the interface. If no return type is specified in the interface, the implementing methods can declare any return type.

Getter and setter functions work differently in interfaces. The special syntax for function `getPropertyName()` is not supported.

CHAPTER 12

Working with ActionScript in Flex

Macromedia Flex provides you with a way to build applications with a set of MXML tags, however, you also use ActionScript to perform certain actions with the Flex components. You use ActionScript to define custom functions and methods, call ActionScript functions, and work with components.

This chapter describes how to perform these actions using ActionScript in your MXML applications.

Contents

| | |
|---|-----|
| Using ActionScript in Flex | 319 |
| Working with components | 320 |
| About scope | 325 |
| Using the <code>doLater()</code> method | 333 |

Using ActionScript in Flex

Macromedia Flex provides a means of building applications with a set of MXML tags, but you also need ActionScript to perform certain actions with the components that are represented by those tags.

You use ActionScript in your Flex applications to do the following:

- Define custom functions and methods
- Access document and application scopes
- Call ActionScript functions
- Create and destroy components

This chapter describes how to perform these actions using ActionScript in your Flex applications. For more basic information about using ActionScript, see *Getting Started with Flex*.

For additional information about using ActionScript in Macromedia Flash development, see *Flex ActionScript Language Reference*.

Working with components

ActionScript blocks in your MXML files or ActionScript classes in external packages provide a powerful way to manipulate your MXML components.

You use scripts to attach properties, methods, and events to the object at the root of the MXML document. In an MXML application, the scripts are attached to the Application object (which corresponds to the `<mx:Application>` tag) or whatever the top-level tag in the current document is. In an MXML component, the scripts are attached to the component object.

This section describes how to work with components and their properties with ActionScript.

Component basics

To work with a component in ActionScript, you must define an `id` property for that component in the MXML tag. This property is optional if you do not want to access the component with ActionScript.

For example, the following code sets the `id` property of the Button control to `myButton`:

```
<mx:Button id="myButton" label="Click Me" />
```

After you define the Button control's `id` property, you can explicitly refer to this Button control's instance with its `id` property in any ActionScript class or script block. By referring to a component's instance, you can modify its properties and call its methods.

For example, the following ActionScript block changes the value of the Button control's `label` property when the user clicks the button:

```
<mx:Button id="myButton" label="Click Me" click="setLabel();" />
<mx:Script><![CDATA[
    function setLabel() {
        myButton.label = "Click";
    }
]]></mx:Script>
```

MXML applications have a flat namespace. This means that all `id` properties must be unique within a document. It also means that you can address any `id` property from anywhere in the application: functions, external class files, imported ActionScript files, or inline scripts. You can think of an `id` property as a public variable, so in order to access it, you must reference the class in which it occurs.

Calling component methods

You can invoke the public methods of a component instance in your Flex application using the following dot-notation syntax:

```
componentInstance.method([parameters]);
```

The following example invokes the `sortList()` method of the `list1` List control when the user clicks the button, which invokes the public `sortItems()` method of the List control:

```
<mx:List id="list1" initialize="list1_init();" />
<mx:Script><![CDATA[
    function sortList(list_obj) {
```



```

        list_obj.sortItems();
    }
    function list1_init() {
        // Populate list with values.
    }
}]></mx:Script>
<mx:Button id='b' label="Sort List 1" click="sortList(list1);" />

```

To invoke a method from a child document (such as a custom MXML component), you can use the `parentApplication`, `parentDocument`, or `Application.application` properties. For more information, see [“About Document and Application scopes” on page 329](#).

Initializing components

Every component supports an `initialize` event, which lets you define actions that occur before Flex draws the component on the screen. To initialize a component with ActionScript, set the `initialize` property to point to a function that you create in an `<mx:Script>` block.

Note: Because Flex invokes the `initialize` event before drawing the component, you cannot access size and position information of that component from within the `initialize` event handler unless you use the `creationComplete` event handler. For more information on the order of initialization events, see [“About component startup order” on page 348](#).

The following example points the `initialize` property to the `initDate` function. When Flex finishes instantiating the `Label` control and before it draws the first window of the application, Flex calls the `initDate` function.

```

<mx:Label id="label1" text = "Today's Date: " initialize="initDate();"
    width="300" />
<mx:Script><![CDATA[
    function initDate() {
        label1.text = label1.text + Date();
    }
}]></mx:Script>

```

You can also express the previous example without an explicit function call by adding the ActionScript code in the component’s inline definition, as the following example shows:

```

<mx:Label id="label1" text = "Today's Date: " initialize="label1.text =
    label1.text + Date();" width="300" />

```

As with other calls that are embedded within component definitions, you can add multiple ActionScript statements to the `initialize` property by separating each function or method call with a semicolon. You can pass values to the functions just as you would with any other function call.

The following example calls both the `initDate` and the `changeColor` functions when the `label1` component is instantiated:

```

<mx:Label id="label1" text = "Today's Date: "
    initialize="initDate();setColor('blue');" width="300" />
<mx:Script><![CDATA[
    function initDate() {
        // function implementation
    }
    function setColor(String:c) {

```

```

    // function implementation
}
]]></mx:Script>

```

To avoid complicating the tag definition, you should generally create a separate `ActionScript` function that handles multiple statements, rather than adding them inline.

Using component properties

Functions in `ActionScript` blocks can read and write component properties. You use the following dot-notation to specify a property:

```
componentInstance.property = value;
```

The following example changes the `label` property of the `Button` control when the user clicks the button:

```

<mx:Button id='b' label="Change Label" click="changeLabel(b,'New');" />
<mx:Script><![CDATA[
    function changeLabel(obj, lbl) {
        obj.label = lbl;
    }
]]></mx:Script>

```

Adding component properties

Functions and statements in `ActionScript` blocks can define new properties for the top-level component of the current file. For example, adding a variable declaration in your root MXML file adds a new property to the application object. You can then access this property from other `ActionScript` functions.

To see all properties of the application object, you can iterate over it using the following sample code:

```

<mx:Script><![CDATA[
    function getAppProps() {
        var fl:String="";
        for (var foo in application) {
            fl = fl + "application." + foo + " = " + application[foo] + "\n";
        }
        return fl;
    }
]]></mx:Script>

```

As with any variable declared in `ActionScript`, the property is a member variable of the class. For more information on scope in `ActionScript`, see [“About scope” on page 325](#).

The following example creates a new property, `count`, and increments its value each time the user clicks the button:

```

<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
    <mx:Script><![CDATA[
        var count:Number = 0;
        public function getCount():Number {
            return count++;
        }
    ]]]>

```

```

]]></mx:Script>
<mx:Button id="b1" label="Get Count" click="taCount.text=getCount();" />
<mx:TextArea id="taCount" />
</mx:Application>

```

If you are executing `ActionScript` code within a child document, such as a custom `MXML` component, you can access the application object's properties using the `parentApplication` or `Application.application` properties. For more information, see [“About Document and Application scopes” on page 329](#).

Object property introspection

You can use a simple `for in` loop in `ActionScript` to view all properties of an object. The following sample function takes an object as an argument and traces all the properties of that object:

```

function dumpObj ( obj ) {
    for (var i in obj) {
        trace (i + " : " + obj[i]);
    }
}

```

You can also output the trace statements to a `TextArea` in your Flex applications, as the following example shows:

```

<?xml version="1.0" encoding="iso-8859-1"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
    <mx:Script><![CDATA[
        function dumpObj ( obj ) {
            for (var i in obj) {
                ta1.text = ta1.text + i + " : " + obj[i] + "\n";
            }
        }
    ]]></mx:Script>
    <mx:TextArea id="ta1" width="400" height="500" />
    <mx:Button label="Introspect Me" click="dumpObj(this)" />
</mx:Application>

```

Instantiating components in `ActionScript`

In some cases, you defer the instantiation of components and want to create them at a later time. In other cases, you do not declare the component in `MXML` at all and want to dynamically create them in a `<mx:Script>` block. This section describes how to instantiate components in `ActionScript`.

You must have an import or reference to the component you want to create in `ActionScript`. This is because when the Flex application is compiled, it must already have a reference to the class that will be instantiated later.

Creating components not declared in `MXML`

If you want to create a new child component that was not declared in `MXML`, use the `createChild()` method; for example:

```

<mx:Script>
    import mx.controls.Button;

```

```

    var okButton:Button;
    function someMethod():Void {
        okButton = Button(createChild(Button, undefined, { label: "OK" }));
    }
</mx:Script>

```

The `createChild()` method is described in detail in [“Creating component instances at runtime” on page 182](#).

Flex creates the new child as the last child in the container. If you do not want the new child to be the last in the container, use the `setChildIndex()` method to change the order.

You should pass `undefined` for the second parameter, which is the instance name (and is different from a Flex id). Store the resulting `UIObject` reference into a strongly-typed script variable; to do this you cast the result of `createChild()`.

You can call `createChild()` as many times as you want, but if you call it more than once you should store the `UIObject` references in different variables. For example, you could declare a script variable which is an `Array` and store the references into `myDynamicChildren[i]`.

Creating deferred components

In some cases you add a component in MXML but defer its instantiation for a later time (if at all). You might do this to improve initial layout performance or you might want to make the use of some components conditional. You do this by setting the `creationPolicy` property to `none` on the component and then calling the `createComponent()` method.

The following example creates a `Button` control with the `createComponent()` method, but only after the `createButton()` method is called:

```

<mx:Script><![CDATA[
    function createButton():Void {
        p1.createComponent(myDeferredButton);
    }
]]></mx:Script>
<mx:Panel id="p1">
    <mx:Button id="myDeferredButton" label="Go" creationPolicy="none" />
</mx:Panel>

```

You can only instantiate the deferred component once. If you call the `createComponent()` method again, Flex does not create another instance of the component.

For more information about the `createComponent()` method and setting the `creationPolicy` property, see [“Using the createComponent\(\) method” on page 582](#).

Implementing interfaces

MXML components can implement interfaces, as the following example shows:

```

<mx:HBox xmlns:mx="http://www.macromedia.com/2003/mxml"
    implements="MyInterface">

```

You then implement the methods of the interface in your `<mx:Script>` block.

You can implement multiple interfaces by adding a comma-separated list of interfaces to the `implements` property, as the following example shows:

```
<mx:HBox xmlns:mx="http://www.macromedia.com/2003/mxml"
  implements="MyComponentPanelInterface, MyComponentOtherInterface">
```

For more information about using interfaces in MXML files, see [“Using interfaces” on page 317](#).

You write interfaces in ActionScript. For more information about writing interface files, see *Flex ActionScript Language Reference*.

About scope

Scoping in ActionScript is largely a description of what the `this` keyword refers to at any given point. ActionScript lets you pass functions and call those functions using different scopes, so the `this` keyword can refer to different objects at different times. In general, an object’s methods execute within the context of the object and not the calling object.

In your application’s core MXML file, you can access the `Application` object using the `this` keyword. However, in custom ActionScript and MXML components, event handler objects, or external ActionScript class files, Flex executes in the context of those objects and classes, and the `this` keyword refers to the current scope and not the `Application` scope.

You can access the parent `Document` and `Application` objects using the following methods:

- Reference the `parentDocument`, `parentApplication`, or `Application.application` properties.
- Pass a function reference to the target object.

Flex includes an `Application.application` property that you can use to access the root application. In addition, you can use the `parentDocument` property to access the next level up in the document chain of a Flex application, or the `parentApplication` property to access the next level up in the application chain.

Scoping in event handlers can be confusing. Event handlers that are declared as objects execute in their own scope and not in the scope of the object that triggered the event or the application. As a result, you can use the `Delegate` class to pass a reference to the proper scope when defining an event handler.

The following sections discuss function scope, variable scope, reserved words, the `this` keyword, and using the `Application.application`, `parentApplication`, and `parentDocument` properties to access scopes outside of the current context. For more information about event handler scoping, see [“Scoping in event handlers” on page 342](#).

Scoping functions

In ActionScript, you can pass a function from one object to another. When the second object invokes the function, the function runs in the context of the second object. Sometimes you must run the function in the scope of the original object, such as when you define an event handler as an object.

The following example defines a new object with a single property and a single method:

```
var myObj = new Object();
myObj.color = "red";
myObj.setColor = function(c) {
    this.color = c;
}
```

You can pass `myObj` to a method and it can call the `myObj.setColor("green")` method. In this case, `this` refers to `myObj` and `this.color` is the `color` property of `myObj`, as you would expect. However, in `ActionScript` you can assign a function to a property of a second object, as the following example shows:

```
var myOtherObj = new Object();
myOtherObj.setOtherColor = myObj.setColor;
```

In this example, in a call to the `myOtherObj.setOtherColor()` method, `this` refers to `myOtherObj`. If you have logic in the `myObj.setColor()` method that assumes that `this` refers to `myObj`, unexpected results can occur.

`ActionScript` lets you pass the scope of a function from one object to another. Some global functions in `ActionScript` even take scope arguments, such as `setInterval()`.

`Flex` provides a `Delegate` object that you can use to wrap a function with a particular scope object. When you call the function, it executes in the scope that you intended.

The `Delegate` utility class is in the `mx.utils` package. You create an instance of the `Delegate` utility class with the `create()` method, as the following example shows:

```
mx.utils.Delegate.create(scope, function)
```

The `Delegate` class ensures that the function meant to run in `myObj`'s scope continues to run in that scope. The following code shows how you would use the `Delegate` class to do this:

```
myOtherObj.setOtherColor = mx.utils.Delegate.create(myObj, myObj.setColor);
```

With the `Delegate` class, calling the `myOtherObj.setOtherColor("green")` method is the same as calling the `myObj.setColor("green")` method.

The `Delegate` utility class is especially useful when including per-instance event handlers on web service calls. For example, you can write functions that you want called when the service returns or faults, such as the following:

```
function myResultHandler(result) {
    this.myResultLabel.text = result.value;
}
function myFaultHandler(fault) {
    this.alert('There was a problem.');
```

Without the `Delegate` utility class, your event handlers might resemble the following:

```
var call = myService.someMethod();
call.onResult = myResultHandler;
call.onFault = myFaultHandler;
```

When you run this, Flex does not set the value of the text label or raise the Alert; Flex calls the functions themselves. In this case, you add a Delegate class to execute functions in the proper scope, as the following code shows:

```
import mx.utils.*;
var call = myService.someMethod();
call.onResult = Delegate.create(this, myResultHandler);
call.onFault = Delegate.create(this, myFaultHandler);
```

If you cannot determine what the `this` keyword refers to in a function, you can add a breakpoint using the fdb debugger utility. In the breakpoint, trace the value of `this` with a statement like `"print *this"`. For more information, see [“Using the debugger” on page 757](#).

Variable scope

Variables declared within a function are locally scoped to that function. These variables can share the same name as variables in outer scopes, and they do not affect the outer-scoped variable. General programming practices advise that you scope variables locally. You can refer to the outer-scoped variable with the `this.` prefix.

In the following example, the value of `xxx` is set in the outer scope to 5. The `doSomething()` method called during initialization defines a locally scoped variable of the same name and sets it to 10. When the user clicks the button, the `doSomethingElse()` method sets the value of the `TextArea` to the value of the outer-scoped variable.

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
  initialize="doSomething()">
  <mx:Script><![CDATA[
    var xxx = 5;
    function doSomething() { //On initialization, sets value of TextArea to 10.
      var xxx = 10;
      tal.text = xxx;
      trace(this.xxx); // Traces 5, since this.xxx refers to the outer scope.
    }
    function doSomethingElse() {
      tal.text = xxx;
    }
  ]]></mx:Script>
  <mx:VBox>
    <mx:TextArea text="" id="tal"/>
    <mx:Button id="b1" label="Do Something Else" click="doSomethingElse();"/>
  </mx:VBox>
</mx:Application>
```

If you remove the variable declaration from the `doSomething()` method, the function sets the value of the outer variable `xxx` to 10, as the following code shows:

```
function doSomething() { //On initialization, sets value of TextArea to 10.
  xxx = 10;
  tal.text = xxx;
}
```

Variables, methods, and functions in classes, interfaces, and included files used in the MXML file are available in the application scope. The following example shows the contents of two files. The first file, `myApp.mxml`, calls the `getCount()` function, which is in the second file, `myInclude.as`.

The following example shows the contents of the `myApp.mxml` file:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:Script source="myInclude.as" />
  <mx:Button id="b1" label="Compute Sum" click="taCount.text=getCount();" />
  <mx:TextArea id="taCount" />
</mx:Application>
```

The following example shows the contents of the `myInclude.as` file:

```
var count:Number = 0;
public function getCount():Number {
    return count++;
}
```

Using the `this` keyword

The `this` keyword refers to the object in the currently executing scope. From inside an object method, use the `this` keyword to reference the object instance. You can also use the `this` keyword to pass a class instance as an argument of a method, to itself or another class. This is a common way of setting up a callback mechanism in your classes.

In an `<mx:Script>` tag, the `this` keyword refers to the application object, or the top-level component. If you are executing a method in an `ActionScript` or `MXML` component, the `this` keyword refers to that component and not the application root.

The following example shows accessing the application object:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:Script><![CDATA[
    function startup() {
      this.loginPanel.setVisible(true);
      this.loginPanel.setModel(true);
    }
  ]]></mx:Script>
  <mx:Window id="loginPanel" title="Login" width="250" height="150"
    visible="false" >
    <mx:VBox verticalAlign="top" marginRight="8" marginLeft="8" marginTop="8"
      marginBottom="8">
      <mx:HBox horizontalAlign="center">
        <mx:Button id="b1" label="Login" click="this.startup();" />
        <mx:Button id="b2" label="Cancel"
          click="this.loginPanel.setVisible(false);" />
      </mx:HBox>
    </mx:VBox>
  </mx:Window>
</mx:Application>
```


One common use for the `this` keyword is to allow access to a field variable when there is a local variable with the same name in a different scope. The following example shows how using the `this` keyword can avoid confusion, because `foo` is a local variable and a property of the class:

```
<mx:Script><![CDATA[
    var foo:String;
    function setFoo(foo:String) : Void {
        this.foo = foo;
    }
]]></mx:Script>
```

In class definitions, you do not have to refer to member variables and functions using the `this` keyword, but you can.

As with using `this` in `<mx:Script>` blocks, using the `this` keyword in any event handler points to the top-level object in the MXML file that contains that event handler. It does not point to the event object or to the object that triggered the event, unless that is the top-level object.

In an `ActionScript` class file, `this` refers to the class defined by that file. In the following example, the `this` keyword refers to an instance of `myClass`. Because `this` is implicit, you do not need to include it.

```
class myClass {
    var _x:Number = 3;
    function get x():Number {
        return this._x;
    }
    function set x(y:Number):Void {
        if (y > 0) {
            this._x = y;
        } else {
            this._x = 0;
        }
    }
}
```

About Document and Application scopes

Flex applications consist of multiple documents and multiple applications. The following sections describe the Document and Application objects, and how to use properties to access their scopes.

About the Application object

Flex defines any MXML file that contains an `<mx:Application>` tag as an Application object. In most cases, your Flex application has one Application object, which is the root document of the application. Some applications use the Loader control to add additional applications.

An Application object has the following characteristics:

- Application objects are MXML files with an `<mx:Application>` tag.
- Most Flex applications have a single Application object.
- The root Application is the first application that is loaded.

- An Application object is also a Document object, but a Document object is not always an Application object.
- You can refer to the root Application as `mx.core.Application.application` from anywhere in the Flex application.
- If you load multiple nested applications using the Loader control, you can access the scope of each one from the bottom up using `parentApplication`, `parentApplication.parentApplication`, and so on.

About the Document object

Flex creates a Document object for every MXML file used in a Flex application. For example, you can have a root document, which is also an Application object, and from there, use other MXML files that define custom controls.

A Document object has the following characteristics:

- All *.mxml files used by a Flex application are Document objects, including the root Application's document.
- Custom ActionScript component files are Document objects.
- You cannot directly request documents that are not Application objects. Macromedia Flash Player cannot compile a SWF file from a file that does not contain an `<mx:Application>` tag.
- Documents usually consist of MXML custom controls that you use in your Flex application.
- You can access the parent document's scope using `parentDocument`, `parentDocument.parentDocument`, and so on.
- Flex provides an `isDocument()` method so that you can detect if any given object is a Document object.

Accessing Document and Application scopes

In your application's core MXML file, you can access the methods and properties of the Application object using the `this` keyword. However, in custom ActionScript and MXML components, event handler objects, or external ActionScript class files, Flex executes in the context of those objects and classes, and the `this` keyword refers to the current Document object and not the Application object. You cannot refer to a control or method in the application from one of these child documents without specifying the location of the parent document.

Flex provides the following properties that you can use to access parent documents:

- **`mx.core.Application.application`** The top-level application, regardless of where in the document tree your object executes.
- **`parentDocument`** The parent document of the current document. You can use `parentDocument.parentDocument` to walk up the tree of multiple documents.
- **`parentApplication`** The application object in which the current object exists. Since Flex applications can load applications into applications, you can access the immediate parent application using this property. You can use `parentApplication.parentApplication` to walk up the tree of multiple applications.

The following sections describe these properties.

Using the `mx.core.Application.application` property

To access properties and methods of the top-level application, you can use the `application` property of `mx.core.Application`. This property provides a reference to the `Application` object from anywhere in your Flex application.

You can use the `application` property in a function or in the tag of an MXML component:

```
<mx:Button click="mx.core.Application.application.doSomething();" />
```

Rather than use the full package name when referring to the members of the `Application` class, you can import the package at the top of the class, as the following example shows:

```
import mx.core.Application;
function B1_initialize(event) {
    ...
    Application.application.B1.label="myButton";
}
```

The `application` property is especially useful in applications that have one or more custom MXML or `ActionScript` components that each use a shared set of data. At the application level, you often need to store information and provide utility functions that any of the components can access.

For example, suppose that you store the user's name at the application level and you implement a utility function, `getSalutation()`, which returns the string "Hi, *username*". The following example `MyApplication.mxml` file shows the application source that defines the `getSalutation()` method:

```
<mx:Script>
    var userName:String;
    function getSalutation() {
        return "Hi, " + userName.substring(0, userName.indexOf(" "));
    }
</mx:Script>
```

The `<mx:Script>` tag contents, event handlers, and the bindings of a component execute in the context of that component and not the context of the application. To access the `userName` and call the `getSalutation()` method in your MXML components, you can use the `application` property, as the following example code from the `MyComponent.mxml` component shows:

```
<mx:Script>
    function doOneThing() {
        doAnotherThing(mx.core.Application.application.name);
    }
</mx:Script>

<mx:VBox>
    <mx:Label text="{mx.core.Application.application.getSalutation()}" />
    ...
</mx:VBox>
```

The `Alert` static function presents a special case. You cannot invoke it using the `parentApplication` property. Instead, you must use the full package name if you want an `Alert` to show up from inside a custom component.

The following MXML file uses the `myAccordion` custom component:

```
<mx:Application xmlns:my="*" xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:TabNavigator width="100%" height="100%" id="tnTop">
    <my:myAccordion/>
  </mx:TabNavigator>
</mx:Application>
```

The following custom component (`myAccordion.mxml`) displays an `Alert` when a user clicks the button:

```
<mx:Accordion xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:VBox width="100%" height="100%" label="VBox" id="vb1">
    <mx:Button label="Button" click="mx.controls.Alert.show('Alert!');"/>
  </mx:VBox>
</mx:Accordion>
```

Using the `parentDocument` property

To access the parent `Document` of an object, you can use the `parentDocument` property. All classes that inherit from the `UIObject` class have a `parentDocument` property.

The following example from a custom MXML component uses the `parentDocument` property to define an `Accordion` container that is slightly smaller than the enclosing container:

```
<mx:Accordion width="{parentDocument.width*.80}"
  height="{parentDocument.height*.50}" label="Accordion"
  xmlns:mx="http://www.macromedia.com/2003/mxml">
```

You use the `parentDocument` property in MXML scripts to go up a level in the chain of document objects. You can use the `parentDocument` to walk this chain using multiple `parentDocument` properties, as the following example shows:

```
parentDocument.parentDocument.doSomething();
```

The `parentDocument` property of the root `Application` object is not a reference to itself. For the `Application` object, `parentDocument` is undefined.

The `parentDocument` is typed as an `Object` so that you can access properties and methods on ancestor `Document` objects without casting.

Every `UIObject` class has an `isDocument()` method that returns `true` if that `UIObject` class is a document object and `false` if it is not. If a `UIObject` class is a document object, it has a `documentDescriptor` property. This is a reference to the descriptor at the top of the generated descriptor tree in the generated document class.

For example, suppose that `AddressForm.mxml` creates a subclass of the `<mx:Form>` to define an address form, and `MyApp.mxml` creates two instances of it: `<AddressForm id="shipping">` and `<AddressForm id="billing">`.

In this case, the `shipping` object is a `Document` object. Its `documentDescriptor` property corresponds to the `<mx:Form>` tag at the top of the `AddressForm.mxml` file (the definition of the component), while its descriptor corresponds to the `<AddressForm id="shipping">` tag in `MyApp.mxml` file (an instance of the component).

Walking the document chain using the `parentDocument` property is similar to walking the document chain using the `parentApplication` property.

Using the `parentApplication` property

Applications can load other applications, therefore, you can have a hierarchy of applications, similar to the hierarchy of documents within each application. Every `UIObject` class has a `parentApplication` read-only property that references the `Application` object in which the object exists. The `parentApplication` property of an `Application` object is never itself; it is either the `Application` object into which it was loaded, or undefined (for the root `Application` object).

Walking the application chain using the `parentApplication` property is similar to walking the document chain using the `parentDocument` property.

Using the `doLater()` method

The `doLater()` method queues a function to be called when the selected operation finishes. The `doLater()` method is useful if you have functions or dynamically created controls that rely on nonsequential operations such as web services. Rather than try to time the return of data from a web service call, you can use the `doLater()` method to ensure that the necessary data is available before continuing.

The `doLater()` method is from the `mx.core.UIObject` class. The `doLater()` method has the following signature:

```
doLater(obj:Object, func: String, args: Array):Void
```

The `obj` argument is the object that contains the function. The `func` argument is the function to call on the object. The `args` argument is an optional array of arguments you can pass to the function.

The following example uses a call to the `doLater()` method to ensure that the current operation is completed before the `createNext()` custom method is called:

```
<?xml version="1.0">
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" xmlns:local="*"
  creationComplete="doLater(this,'createNext')">
  <mx:Script><![CDATA[
    var creationOrder = ["box1","box2","box3"];
    var creationIndex = 0;
    function createNext() {
      var nextObj = this[creationOrder[creationIndex++]];
      nextObj.addEventListener("childrenCreated", this);
      nextObj.createComponents();
      if (creationIndex < creationOrder.length) {
        nextObj.getChildAt(0).addEventListener("creationComplete", this);
      }
    }
    function handleEvent(e:Object):Void {
      if (e.type == "creationComplete") {
        doLater(this, "createNext");
      } else {
        super.handleEvent(e);
      }
    }
  ]]>
</mx:Script>
</mx:Application>
```

```

    }
  ]]></mx:Script>
  <mx:VBox id="box1" width="200" height="200" creationPolicy="none">
    <mx:DataGrid width="190" height="95">
      ... // Add data provider here.
    </mx:DataGrid>
  </mx:VBox>
  <mx:VBox id="box2" width="200" height="200" creationPolicy="none">
    <mx:DataGrid width="190" height="95">
      ... // Add data provider here.
    </mx:DataGrid>
  </mx:VBox>
  <mx:VBox id="box3" width="200" height="200" creationPolicy="none">
    <mx:DataGrid width="190" height="95">
      ... // Add data provider here.
    </mx:DataGrid>
  </mx:VBox>
</mx:Application>

```

CHAPTER 13

Using Events

One of the most important parts of your Macromedia Flex application is handling events. This chapter describes how to handle events using controls and ActionScript in your Flex applications.

Contents

| | |
|-----------------------------------|-----|
| About events | 335 |
| Handling events | 337 |
| Handling mouse events | 352 |
| Using base class events | 353 |

About events

Flex applications are event-driven. Events let a programmer know when the user interacts with the interface, and also when important changes happen in the appearance or life cycle of a component, such as the creation or destruction of a component or its resizing.

Events can be generated by user input devices, such as the mouse and keyboard, or by external forces, such as the return of a web service call.

Each component has built-in events that you handle in ActionScript blocks in your MXML applications. In addition, you can take advantage of the Flex event system's dispatcher-listener model to define your own event handlers outside of your applications, and define which objects listen to certain events. You can register listeners with the target object so that when the target object broadcasts an event, the listeners get called.

Components generate and broadcast events and *consume* (listen to) other events. An object that requires information about another object's events registers with that object. When an event occurs, the object broadcasts the event to all registered listeners by calling a function requested during registration. To receive multiple events from the same object, you must register for each event.

Note: The Flex event model does not currently support capture and bubbling.

For a description of each component's events, see the component's description in [Chapter 2](#), "Using Controls," on page 31.

Using the event object

The *event object* is an ActionScript object with properties that contain information about the event that occurred. The event object is an implicitly created object in MXML, much the same way the session, request, and response objects are implicitly created by the application server in a JavaServer Page (JSP). When you use the event object in an MXML tag, you must refer to it as event. You pass it to an event handler as a parameter. You must pass the event object to an event handler only if you use properties of the event object in the handler.

An *event handler* is a function that responds to events, often by accessing the properties of the event object or some other settings of the application state. Event handlers can only change the state of some object. The following example shows a simple event handler function that clears a TextArea control when the user clicks the button:

```
<mx:Button label="Clear" click="clickEvt()" >
<mx:TextArea id="ta1" width="150" text="This will be cleared" />
<mx:Script><![CDATA[
    function clickEvt() {
        ta1.text="";
    }
]]></mx:Script>
```

Passing an event object to, and using it in, an event handler is optional. However, if you want to access the event object's properties inside your event handlers, you must pass the event object to the handler. You can use the event object inside the handler to access details about the event that was broadcast, or about the component that broadcast the event. With a reference to the instance name of the broadcasting component, you can access all the properties and methods of that instance.

The following example creates two event handler functions and registers them with the events of a ComboBox control. The first event handler, `openEvt()`, takes no arguments. The second event handler, `changeEvt()`, takes the event object as an argument and uses this object to access the value and `selectedIndex` of the ComboBox control that triggered the event.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
    <mx:Script><![CDATA[
        function openEvt() {
            forChange.text="";
        }
        function changeEvt(e) {
            forChange.text=e.target.value + " " + " " + e.target.selectedIndex;
        }
    ]]></mx:Script>
    <mx:ComboBox open="openEvt()" change="changeEvt(event)" >
        <mx:dataProvider>
            <mx:Array>
                <mx:String>AK</mx:String>
                <mx:String>AL</mx:String>
                <mx:String>AR</mx:String>
            </mx:Array>
        </mx:dataProvider>
    </mx:ComboBox>
```



```
<mx:TextArea id="forChange" width="150" />
</mx:Application>
```

Event object properties

An event object is created each time an event is triggered. Depending on the component that triggered the event and the event itself, the event object can have a wide range of properties. These properties are based on those defined in the W3C specification (www.w3.org/TR/DOM-Level-3-Events/events.html), but Flex does not implement all of these.

The following table describes the properties for all events of the event object:

| Property | Type | Description |
|----------|--------|--|
| type | String | The name of the event; for example, <i>click</i> . |
| target | Object | A reference to the component instance that broadcast the event. You cannot change this to a different object. You access the name of the object using the <code>target.id</code> property. |

The description of each event in the Flash MX 2004 documentation lists the event properties that are optional and required. For example, the `ScrollBar.scroll` event adds a `detail` property in addition to the `type` and `target` properties. For more information, see the event descriptions in the Flash MX 2004 documentation.

Event objects generated by Flex for keyboard and mouse events have their own set of properties. These properties let you determine relevant information about the event, such as the key or sequence of keys that were pressed or the position of the mouse when the event was emitted. For more information, see “[Handling mouse events](#)” on page 352 and “[Using base class events](#)” on page 353.

Handling events

When a component raises an event, two groups of objects are notified, in the following order:

1. Instances of the component
2. Objects that have registered as listeners for that event

There are several different strategies that you can employ when handling events:

- Define an event handler inline. This binds a call to the event handler to the control that triggers the event:

```
<mx:Button label="Click Me" click="myEventHandler();" />
```

In this example, whenever the user clicks the button, Flex calls the `myEventHandler()` function. For more information, see “[Defining event handlers inline](#)” on page 338.

- Define an event listener and register components to call the listener object’s functions using the `addEventListener()` function, as the following example shows:

```
<mx:Script><![CDATA[
    function createListener() {
        var myListener = new Object();
        myListener.click = function() {
```

```

        //Handle the event.
    }
    b1.addEventListener("click", myListener);
}
]]></mx:Script>
<mx:Button label="Click Me" id="b1" />

```

In this example, whenever the user clicks the button, Flex calls the listener's `click` handler function. You can register multiple components with this event handler, or add multiple listeners to a single component. For more information, see [“Using event listeners” on page 340](#).

- Create an event listener class and register components to use that class for event handling. This approach of event handling promotes code reuse and lets you centralize event handling outside of your MXML files. For more information, see [“Creating event listener classes” on page 347](#).

The following sections describe these methods of handling events.

Defining event handlers inline

The simplest method of handling events in Flex applications is to point to an event handler function in the component's MXML tag. To do this, you add any of the component's events as a tag property followed by an `JavaScript` statement or function call.

Add an event handler inline using the following syntax:

```
<mx:TagName eventName="handlerFunction" />
```

For example, to handle a `Button` control's `click` event, you point the `<mx:Button>` tag's `click` property to a function, and then define that function in an `JavaScript` block. The following example defines the `submitForm()` function as the handler for the `Button` control's `click` event:

```

<mx:Button label="Submit" click="submitForm();" />
<mx:Script><![CDATA[
    function submitForm() {
        // Do something.
    }
]]></mx:Script>

```

Passing parameters to event handlers

You can pass parameters to the event handler function. The following example passes the event object to the `submitForm()` event handler:

```

<mx:Button label="Submit" click='submitForm(event);' />
<mx:Script><![CDATA[
    function submitForm(evtObj) {
        // Do something with the event object.
    }
]]></mx:Script>

```

Defining multiple event handlers

You can define multiple handler functions for a single event by separating each function call with a semicolon. The following example calls the `submitForm()` and `debugMessage()` functions when the user triggers the `click` event:

```
<mx:Button click='submitForm(event); debugMessage("Debug message");' label="Do Both Actions"/>
```

You can assign a single function to handle multiple events from the same or different components. The following example creates two `Button` controls that move a third `Button` control up or down. Each `Button click` event also calls the `changeLabel()` function that changes the label of the moved `Button` control in the application window:

```
<mx:HBox height="50">
  <mx:Button id='a' label="Button1" width="80" />
</mx:HBox>
<mx:HBox height="300">
  <mx:Button label="Up" click='runMove("up");' width="80"/>
  <mx:Button label="Down" click='runMove("down");' width="80"/>
</mx:HBox>
<mx:Script><![CDATA[
  function runMove(dir) {
    if (dir == "up") {
      a.y=a.y-20;
    } else if (dir == "down") {
      a.y=a.y+20;
    }
    changeLabel(a);
  }
  function changeLabel(but) {
    but.label = String(but._x) + "," + String(but._y);
  }
  ]]></mx:Script>
```

Using ActionScript in event handlers

Event handlers can include any legal ActionScript code, including calling global functions and setting a component property to the return value. The following example calls the `getVersion()` global function and sets the value of `debug.text` to its return value:

```
<mx:TextArea id="debug" width="250" height="50"/>
<mx:Button label="Get Ver" click='debug.text=getVersion();' width="80"/>
```

Using the initialize event handler

The `initialize` property defines the most common inline event handler. Flex calls this event when it instantiates the object, and all Flex components emit an `initialize` event. You can use it to set the styles or data values of your controls, or trigger the creation of other components. When setting runtime style, you should try to set the styles as properties on controls rather than calling the control's `setStyle()` method. Doing this can give you better performance.

When working with multiview containers, be aware that Flex calls the `initialize` event on each view when it first creates a navigator container, but not on the children within each view of that container.

Note: If the ActionScript in your `initialize` event handler refers to child controls of multiview containers (such as the Accordion container) that Flex has not yet instantiated, Flex attempts to execute the statements on undefined objects. Undesirable results can occur.

You should also be aware that the `initialize` event can occur too soon in the component life cycle for you to access all properties and child components, so you should consider using the `creationComplete` or `childrenCreated` events instead. For more information, see [“About component startup order” on page 348](#).

The following example initializes the values of a DataGrid control:

```
<mx:Script><![CDATA[
    function initData() {
        myGrid.dataProvider = [
            {Name:"Bob", Department:"Sales", Extension:"2345"},
            {Name:"Sue", Department:"Marketing", Extension:"5432"},
            {Name:"Fred", Department:"Engineering", Extension:"1122"},
            {Name:"Betty", Department:"Sales", Extension:"8854"},
            {Name:"Steve", Department:"Marketing", Extension:"2389"},
            {Name:"Marsha", Department:"Engineering", Extension:"9964"}
        ];
    }
]></mx:Script>
<mx:DataGrid id="myGrid" initialize="initData()" label="Mouse Down" />
```

Using event listeners

An event listener listens for events that objects dispatch. Listeners can be objects, functions, or classes.

You declare an event listener as an object and define the events that the listener listens for and subsequently handles. You then call the `addEventListener()` method to register an event with the listener.

Use the following syntax when defining an event listener:

```
var listenerName = new Object();
listenerName.eventName = function([eventObject]) {
    // Handle the event. Optionally access the event object.
}
instanceName.addEventListener("eventName", listenerName);
```

You can call the `addEventListener()` method from any component instance. The syntax for the `addEventListener()` method is as follows:

```
componentInstance.addEventListener(eventName:String, listenerName:Object);
```

The following example defines a new listener object called `myListener`. It then defines the `click` function of the listener, and registers the `click` event of the Button control with that listener. When the user clicks the button, Flex calls the `myListener.click()` function.

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
  initialize="createListener()" >
  <mx:Script><![CDATA[
    function createListener() {
      var myListener = new Object();
      myListener.click = function() {
        // Handle the event.
      }
      b1.addEventListener("click", myListener);
    }
  ]]></mx:Script>
  <mx:Button label="Click Me" id="b1" />
  <mx:TextArea id="forNotes" width="350" />
</mx:Application>

```

Note: In a listener that is defined as an object, the scope is the listener object and not the Document or Application scope. As a result, you cannot access objects, variables, or functions in the Application scope directly; you can only access those defined for the listener. For information on accessing Application scope within an event listener, see [“Scoping in event handlers” on page 342](#).

You can remove an event listener using the `removeEventListener()` method of any component instance. The syntax for the `removeEventListener()` method is as follows:

```
componentInstance.removeEventListener(eventName:String, listenerName:Object)
```

Optionally, event listeners can receive a single argument, the event object. You can use the event object to determine what object called the listener object's functions or to access properties of the control that triggered the event. All event objects have a `target` property and a `type` property, as described in [“Event object properties” on page 337](#).

The following example uses the `id` property of the event object's `target` to determine which Button instance the user clicked:

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
  initialize="createListener()" >
  <mx:Script><![CDATA[
    function createListener() {
      var lo = new Object();
      lo.click = function(event) {
        if (event.target.id == "button1") {
          trace("button 1 was clicked");
        } else if (event.target.id == "button2") {
          trace("button 2 was clicked");
        }
      }
      button1.addEventListener("click", lo);
      button2.addEventListener("click", lo);
    }
  ]]></mx:Script>
  <mx:Button label="Click Me" id="button1" />
  <mx:Button label="Click Me Too" id="button2" />
</mx:Application>

```

Even when you pass the event object to the event handler, you cannot access objects within the Document scope, such as the controls in your MXML application. You must use the Delegate utility class to do this. For more information, see [“Scoping in event handlers” on page 342](#).

Adding event listeners inline

You can add event listeners inline with the component definition. The following Button control definition adds the call to the `addEventListener()` method inline with the Button control's `initialize` property:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
  initialize="createListener()" >
  <mx:Script><![CDATA[
    var myEventListener = new Object();
    function createListener() {
      myEventListener.click = function(event) {
        // Handle click event
      }
    }
  ]]></mx:Script>
  <mx:Button id='b' initialize='b.addEventListener("click",
    myEventListener);' />
</mx:Application>
```

Scoping in event handlers

Within an event handler, you can only access the scope within which the event handler was created with the Delegate utility class. This means that in an event handler, the `this` keyword does not refer to the Application object, but rather the listener object, and you cannot access any component class instances inside the event handler.

For example, if you try to refer to a component by its `id` property in an event listener, Flex does nothing, except in the case of web services. The call to the component actually takes place in the event listener's scope and not the Application scope, as you might expect.

Note: In an event listener, the `this` keyword refers to the listener object, not the Application or Document object.

The second parameter of the `addEventListener()` method can take one of two different types: an event listener object or a function. When you register a function as the event listener, it executes in the context of the object that dispatches the event and not the enclosing document. If that dispatcher is a Button control, the `this` keyword refers to the button and not the application object, as you might expect; the listener can only access objects in the dispatcher's scope.

The following example shows this:

```
<mx:Button id="myButton" initialize="myButton.addEventListener('click',
  this.buttonClickedFunction())"/>
<mx:Button id="myOtherButton" ... />
<mx:Script>
  function buttonClickedFunction(event) {
    // The 'this' keyword refers to myButton instance.
    // You can refer to properties on the myButton instance because you are
```

```

        // in the myButton scope.
        // You cannot refer to the myOtherButton instance.
    }
</mx:Script>

```

In `ActionScript`, you can pass a function from one object to another. When the second object invokes the function, the function runs in the context of the second object. The result is that using the `this` keyword in an object listener refers to the listener and not the `Application` object.

If you register an object as the event listener, the event handling function executes in the context of the listener and not in the context of the document or the dispatching object, as the following example shows:

```

<mx:Button id="myButton" initialize="doSetup()"/>
<mx:Button id="myOtherButton" ... />
<mx:Script>
    var myListener:Object;
    function doSetup() {
        myListener = new Object();
        myListener.click = function(event) {
            // The 'this' keyword refers to myListener instance.
            // You cannot refer to myButton instance.
            // You cannot refer to myOtherButton instance.
        }
        myButton.addEventListener('click', myListener);
    }
</mx:Script>

```

The `Delegate` utility class lets you pass in a function and additionally specify the context in which it should execute. It creates a wrapper for the original function and runs it in whatever context you provide. Creating a `Delegate` class to act as the listener lets you access the `Application` scope and instances of components from within the target's scope.

You create a `Delegate` class using the following static function:

```
mx.utils.Delegate.create(scope, function_to_delegate)
```

For example:

```
var myDelegate:Function = mx.utils.Delegate.create(this, click);
```

Passing the `this` keyword at the time that Flex creates the `Delegate` class provides access to the `Document` scope of the document inside the event listener.

You then use the `addEventListener()` method to register the new `Delegate` with a component:

```
component.addEventListener("event", delegate_name);
```

You can condense these two lines into one:

```
myButton.addEventListener("click",mx.utils.Delegate.create(this,click));
```

The following example uses the `Delegate` class to access the `Document` scope in the event listener:

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    initialize="createListener()" >
    <mx:Script><![CDATA[
        // Required for Delegate.
    ]]>

```

```

import mx.utils.Delegate;
function click(event) {
    tal.text = "This is a log message";
}
function createListener() {
    // The 'this' keyword refers to the Application.
    // You can access the tal instance that is in the document scope.
    // You can access the button1 instance that is in the document scope.
    var myDelegate:Function = Delegate.create(this, click);
    button1.addEventListener("click", myDelegate);
}
]]></mx:Script>
<mx:Button label="Submit" id="button1" />
<mx:TextArea id="tal" text="" width="200" />
</mx:Application>

```

You can create a `Delegate` class inline with the `addEventListener()` method to simplify your code, as the following example shows:

```
button1.addEventListener("click", mx.utils.Delegate.create(this, click));
```

Registering multiple events and components

You can register the same listener with several events of the same component, or events of different components. In the latter case, you should add logic to the event listener that processes the type of event. The target (or object that broadcast the event) of the event is added for you. When you register a single listener to handle the events of multiple components, you must use a separate call to the `addEventListener()` method for each instance.

The following example registers a single listener (`myListener`) to the `click` event of a `Button` control and the `click` event of a `CheckBox` control. To detect what type of object called the event handler, the listener checks the `className` property of the target object.

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    initialize="createListener()" >
    <mx:Script><![CDATA[
        function createListener() {
            var myListener = new Object();
            myListener.click = function (event) {
                if (event.target.className == "Button") {
                    // Perform search.
                } else if (event.target.className == "CheckBox") {
                    // Modify search to include all words.
                }
            }
            button1.addEventListener("click", myListener);
            cb1.addEventListener("click", myListener);
        }
    ]]></mx:Script>
    <mx:Button label="Submit" id="button1" />
    <mx:CheckBox label="All Words" id="cb1" />
    <mx:TextArea id="ta" text="Please enter a search term" width="200" />
</mx:Application>

```


The following example creates a Menu control and a TextArea control. When the user selects an item on the menu, the TextArea displays the selected endpoint value. The example registers multiple events, such as change, menuShow, and menuHide to a single handler (menuShowInfo). By defining the Delegate class as a listener, this example lets you access the Document scope inside the menuShowInfo event handler.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
  initialize="initMenu();">
  <mx:Script><![CDATA[
    import mx.utils.Delegate;
    import mx.controls.Menu;
    var myMenu:Menu;
    /* ---Description of Menu Events To Listen For---
    change: Broadcast when a user selects an enabled menu item of type normal,
      check, or radio.
    menuHide: Broadcast when the entire menu or a submenu closes.
    menuShow: Broadcast when the entire menu or a submenu opens.
    rollOut: Broadcast when the cursor rolls off of a Menu item.
    rollOver: Broadcast when the cursor rolls onto a Menu item. */
    function initMenu() {
      myMenu = Menu.createMenu(null, myDP); //Instantiates Menu.
      myMenu.addEventListener("change", Delegate.create(this, menuShowInfo));
      myMenu.addEventListener("menuShow", Delegate.create(this, menuShowInfo));
      myMenu.addEventListener("menuHide", Delegate.create(this, menuShowInfo));
      myMenu.addEventListener("rollOver", Delegate.create(this, menuShowInfo));
      myMenu.addEventListener("rollOut", Delegate.create(this, menuShowInfo));
    }
    function menuShowInfo(event) {
      var itemLabel = event.menuItem.attributes.label;
      taMenuShow.text = "Label: " + itemLabel;
    }
  ]]></mx:Script>
  <mx:HBox>
    <mx:Button id="showBtn" label="Show Menu" click="myMenu.show();" />
    <mx:TextArea id="taMenuShow" text="" width="300"/>
  </mx:HBox>
  <mx:XML id="myDP">
    <node label="Select One" />
    <node type="separator" />
    <node label="Colors" >
      <node label="Soylent Green" />
      <node label="Light Goldenrod Yellow" />
    </node>
    <node label="Names">
      <node label="Guenter" />
      <node label="Reiner" />
      <node label="Wolfgang" />
    </node>
  </mx:XML>
</mx:Application>
```

Registering multiple event listeners for one component

You can register multiple-event listeners to a single component instance, but you must use a separate call to the `addEventListener()` method for each listener. Flex calls each listener function in no specific order.

The following example creates and registers two listeners for one Button control, maintaining the default scope of the listener:

```
<mx:Script><![CDATA[
    function createListener() {
        var myLoggerListener = new Object();
        var myProcessInputListener = new Object();
        myLoggerListener.click = function (event) {
            // ...
        }
        myProcessInputListener.click = function (event) {
            // ...
        }
        button1.addEventListener("click", myLoggerListener);
        button1.addEventListener("click", myProcessInputListener);
    }
}]></mx:Script>
<mx:Button label="Submit" id="button1" />
```

Defining the `handleEvent()` method

You can define a single `handleEvent()` method for the event handler to catch all events.

When you use the `handleEvent()` method in your event listener, you must include logic to determine which target triggered an event. These additional statements add processing overhead, but defining a single event handler function can simplify the coding process.

The following example checks the properties of the event object to determine what to do when an event is triggered. If the event is unrecognized, it logs a message.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    initialize="createListener()" >
    <mx:Script><![CDATA[
        function createListener() {
            var myListener = new Object();
            myListener.handleEvent = function (event) {
                if (event.target.className == "CheckBox") {
                    // Handle click event for CheckBox.
                } else if (event.target.className == "ComboBox") {
                    if (event.type == "open") {
                        // Handle open event of ComboBox.
                    } else if (event.type == "change") {
                        // Handle change event of ComboBox.
                    }
                } else {
                    trace("An event of unexpected type occurred.");
                    trace("Event type: " + event.type);
                    trace("Event target: " + event.target);
                }
            }
        }
    ]></mx:Script>
```

```

    }
}
checkBox1.addEventListener("click", myListener);
comboBox1.addEventListener("change", myListener);
comboBox1.addEventListener("open", myListener);
}
]]></mx:Script>
<mx:CheckBox label="US Territory" id="checkBox1" />
<mx:ComboBox id="comboBox1" >
    <mx:dataProvider>
        <mx:Array>
            <mx:String>AK</mx:String>
            <mx:String>AL</mx:String>
            <mx:String>AR</mx:String>
        </mx:Array>
    </mx:dataProvider>
</mx:ComboBox>
</mx:Application>

```

The following example of an external ActionScript class extends an existing class and adds the ability to capture PAGE-UP and PAGE-DOWN key events:

```

class PagingList extends List {
    var PAGE_DOWN = 120;
    var PAGE_UP = 121;
    function handleEvent(eventObj):Void {
        super.handleEvent(eventObj);
        if (eventObj.type == "KeyDown") {
            if (eventObj.keycode == PAGE_DOWN) vPosition += rowCount;
            else if (eventObj.keycode == PAGE_UP) vPosition -= rowCount;
        }
    }
}

```

Creating event listener classes

You can create external class files to define your event handlers. This lets you use the same event handling logic across applications, which can make your MXML applications more readable. In addition, with event listener classes, the scope that you are executing in can be much clearer and easier to understand.

The following ActionScript class writes a trace message when Flex notifies it of a click event:

```

class MyEventListener {
    function MyEventListener() {
        //Empty constructor.
    }
    function handleEvent(eventObj:Object):Void {
        var type = eventObj.type; // For example, "click"
        var target = eventObj.target.className; // For example, "Button"
        if (type=="click") {
            trace(target + " was " + type + "ed");
        }
    }
}

```

Store your event listener class in `Flex_app_root/WEB-INF/flex/user_classes` or another directory in your `ActionScript` classpath.

The following MXML file instantiates the `MyEventListener` class and defines that class as an event handler for the `click` event:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    initialize="createListener()" >
    <mx:Script><![CDATA[
        function createListener() {
            var myListener = new MyEventListener();
            b1.addEventListener("click", myListener);
        }
    ]]></mx:Script>
    <mx:Button label="Submit" id="b1" />
    <mx:TextArea id="ta1" text="" width="200" />
</mx:Application>
```

About component startup order

All Flex components trigger a number of events during their startup procedure. These events indicate when the component is first created, plotted internally, and drawn on the screen. The events also indicate when the component is finished being created and, in the case of containers, when its children are created.

The following table describes the most commonly used startup events for Flex components:

| Event | Description |
|-------------------------------|---|
| <code>initialize</code> | Broadcast when the component is instantiated. However, you should avoid triggering visual effects on the control until it emits the <code>creationComplete</code> event. |
| <code>childrenCreated</code> | (Containers only) Broadcast after the children of the container are initialized, but before <code>creationComplete</code> is broadcast. When a container emits this event, the container is not yet fully instantiated, but its children are. You can set properties on the children, but not the container itself. To set properties on a container, wait for it to broadcast the <code>creationComplete</code> event. |
| <code>draw</code> | Broadcast when the component is internally drawn. Invisible components <i>do</i> trigger a <code>draw</code> event. |
| <code>creationComplete</code> | Broadcast when the component is measured, laid out, and drawn, but not yet showing on the screen. Containers emit this event only when all of their visible children have been created. |

In addition to these events, the `show` event is triggered during the instantiation of some components. For more information, see [“About show and hide events” on page 350](#).

The following example creates a `VBox` container and `Button` control:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
    <mx:VBox id="child" >
        <mx:Button id="grandChild" />
    </mx:VBox>
</mx:Application>
```

In this example, the Application container is a parent that contains a child VBox container, which contains a child Button control.

The initialization order is as follows:

1. Call `initialize` event on `grandChild`.
2. Call `childrenCreated` event on `child`.
3. Call `initialize` event on `child`.
4. Call `childrenCreated` event on the Application container.
5. Call `initialize` event on the Application container.
6. Call `draw` event on the Application container.
7. Call `draw` event on `child`.
8. Call `draw` event on `grandChild`.
9. Call `creationComplete` event on `grandChild`.
10. Call `creationComplete` event on `child`.
11. Call `creationComplete` event on the Application container.

Knowing the order of startup events lets you identify the dependencies of some controls on other controls. For example, if you have a custom layout algorithm for your container, you could trigger layout to occur after the container's `creationComplete` event, rather than after its `initialize` event.

You must be aware of an object's instantiation life cycle so that you do not write an event handler that references properties of an object that have not yet been set. Properties include the title of a Panel control or the contents of a TextArea control. Flex sets the properties on an object after that object's `childrenCreated` event is triggered, but before Flex draws the object on the screen.

The startup order of multiview containers (navigators) is different from standard containers. By default, all top-level views of the navigator are instantiated. However, only the children of the initially visible view are initialized. When the user navigates to the other views of the navigator, Flex initializes those views' children.

Triggering effects during instantiation

Generally, when you play an effect on an object, you must ensure that the object was created. If you trigger an effect off the `initialize` event, the object's size properties and measuring are not yet complete. The best approach to ensuring that the object is created is to trigger effects off of the `creationComplete` event.

Navigator containers trigger the `changeEffect` event when the currently active view changes. The following example shows the use of the `changeEffect` event:

```
<mx:TabNavigator xmlns:mx="http://www.macromedia.com/2003/mxml"
  xmlns:local="*"
  initialize="addTabs()"
  changeEffect="WipeDown">
```

About show and hide events

Flex components emit two special events when they are made visible or invisible on the screen. These events are triggered only when the `visible` property of a component is set.

The following table describes the `hide` and `show` events:

| Event | Description |
|-------------------|--|
| <code>hide</code> | The component is hidden from view on the screen. Visible components and objects that are invisible at startup time do not trigger a <code>hide</code> event. The <code>hide</code> event is only triggered when a component changes from visible to invisible. |
| <code>show</code> | The component is displayed on the screen. Invisible components and objects that are visible at startup time do not trigger a <code>show</code> event. The <code>show</code> event is only triggered when a component changes from invisible to visible. |

When navigator containers, such as the `ViewStack` and `Accordion` containers, are created, they do not immediately create all of their descendants, but only those descendants that are initially visible. You can instruct Flex to instantiate all children of all containers (regardless of whether they are initially visible or not), or instantiate only select children of select containers. For more information, see [Chapter 24, “Improving Startup Performance,” on page 577](#).

By default, navigator containers fully create the initially visible view, but create only the top-level containers of the hidden views. When the top-level containers that were not initially visible are created, they emit a `creationComplete` event. If you want specific actions to occur when users navigate to a new view, you cannot hook into the navigator’s `creationComplete` event. Instead, you can use the `show` event for the descendants of a navigator container.

The following example creates a `TabNavigator` container with two tabs. The example shows when the `creationComplete` event on a deferred view is emitted, and it shows how to use a `show` event for multiview containers.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    initialize="initApp();">
    <mx:Script>
        var myCount:Number;
        function initApp() {
            myCount = 0;
        }
        function incrementCount() {
            myCount++;
            lbl.text="count: " + myCount;
        }
    </mx:Script>
    <mx:TabNavigator>
        <mx:HBox label="one" show="incrementCount();">
            <mx:Button />
        </mx:HBox>
        <mx:HBox label="two">
            <mx:Button creationComplete="lbl.text='The second pane is complete.'"/>
        </mx:HBox>
    </mx:TabNavigator>
```

```

    <mx:HBox width="200" height="50" borderStyle="solid">
        <mx:Label id="lbl" fontSize="12" />
    </mx:HBox>
</mx:Application>

```

In the previous example, the Label control is empty when the application first starts. When the user browses to the second pane, the second Button control is finally created and it emits the `creationComplete` event. Each time the user returns to the first tab, it emits a `show` event, which increments the counter.

The counter is not incremented on the initial access of the first tab when the application starts because the `show` event requires an explicit change from invisible to visible, which is only attainable by selecting the second tab, and then the first tab.

Manually dispatching events

You can manually dispatch events using a component instance's `dispatchEvent()` method.

When you create an event object and dispatch a new event, you must also define the custom event in metadata outside the class declaration.

The syntax is as follows:

```

[Event("event_name")]
class myClass {
    ...
    dispatchEvent(event_object);
}

```

The Event metadata keyword identifies the event to the ActionScript compiler. You must specify each event with the keyword so that Flex creates a listener for that event.

The syntax for the `dispatchEvent()` method is as follows:

```
dispatchEvent(event_object):Boolean
```

The `dispatchEvent()` method always returns `true`. You can explicitly build an event object before dispatching the event, as the following example shows:

```

var eventObj = new Object();
eventObj.type = "myEvent";
dispatchEvent(eventObj);

```

The event object also has an implicit property, `target`, that is a reference to the object that triggered the event. You cannot change the `target` property, but you can add any number of additional properties to your event object, such as the mouse position or keys that are pressed. Some low-level built-in events capture these properties; for more information, see [“Using base class events” on page 353](#).

You can also use a object initializer syntax that sets the value of the `type` property for the event object and dispatches the event in a single line, as the following example shows:

```
myObject.dispatchEvent({type:"myEvent"});
```

You often use the `dispatchEvent()` method when you create custom components.

Handling mouse events

Macromedia Flash Player can detect a variety of mouse events, including when a user moves their mouse pointer over a component (the `mouseOver` event) and when they click the mouse button (the `mouseDown` event). All components that extend `UIObject` class, including containers and the `Application` object, inherit these events.

All Flex components that inherit from the `UIObject` class support the following mouse events:

- `mouseDown`
- `mouseUp`
- `mouseMove`
- `mouseOver`
- `mouseOut`
- `mouseDownSomewhere`
- `mouseUpSomewhere`
- `mouseChangeSomewhere`
- `mouseMoveSomewhere`

You can define event handlers for the somewhere events to capture global mouse events, while still using control-specific mouse events to handle events.

You can use a `mouseDownSomewhere` event handler on any object, so that it can catch `mouseDown` events outside the object it is on. For example, a `ComboBox` control has an open state and you could add a `mouseDownSomewhere` event handler to close it if the user clicks anywhere in the application.

The global events apply to all controls within the application. For example, if you create a `TabNavigator` container with three `Canvas` containers, and on the first canvas you add the `mouseMoveSomewhere` event, the `mouseMoveSomewhere` event is triggered on all `Canvas` containers in the container.

When Flex generates the event object for mouse events, it does not add any special properties to the object. You can access the `x` and `y` coordinates of the mouse pointer through the `target` property. The event object for this event has the following read-only properties, in addition to the `type` and `target` properties:

```
eventObject.target.mouseX  
eventObject.target.mouseY
```

The `mouseX` and `mouseY` coordinates are relative to the `Application` in the Player and are the mouse pointer's *hotspot* in the coordinate system of the `UIObject`.

A pointer (cursor) is usually larger than 1 pixel by 1 pixel. A particular pixel in the pointer, or hotspot, is placed at the mouse location. This is usually the upper-left corner of the pointer, but the actual location of the hotspot depends on where the registration point is set in the symbol used to create the pointer. It also depends on the offset arguments passed to the `CursorManager.setCursor()` method.

You can specify the `mouseDownSomewhere` event handler on the `Application` tag to capture a mouse click at any time, regardless of where the pointer is.

The following example defines a `mouseDownSomewhere` event handler on the `Application` object:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    mouseDownSomewhere="handleMyMouseClicked()" >
    ...
```

The following example acts as a simple tracker that shows the mouse pointer's current coordinates in the `TextArea` control. When the user moves the mouse pointer over the pixel at (x=200, y=200), the application displays an `Alert` box.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    mouseMoveSomewhere="trackMouse(event)">
    <mx:Script>
    <![CDATA[
    function trackMouse(event) {
        var curX = event.target.mouseX;
        var curY = event.target.mouseY;
        xCoord.text = curX;
        yCoord.text = curY;
        if (curX == 200 && curY == 200) {
            mx.controls.Alert.show("You found it!");
        }
    }
    ]]></mx:Script>
    <mx:TextArea id="xCoord" width="50" />
    <mx:TextArea id="yCoord" width="50" />
</mx:Application>
```

The event object for the `mouseDownSomewhere`, `mouseUpSomewhere`, `mouseChangeSomewhere`, and `mouseMoveSomewhere` events also has a `relatedTarget` property, which is a reference to the component that the mouse pointer is over.

Using base class events

All components derive from the `UIObject` or `UIComponent` base classes. As a result, most components have a common set of low-level events that they can broadcast.

Every event, including the base class events, has a `type` and `target` property, which are explicitly specified by Flex. The event objects for these low-level events sometimes also have unique properties that you can set when using the `dispatchEvent()` method or accessing the event object.

This section describes the low-level events of the `UIObject` and `UIComponent` classes, and the event object properties that are specific to each event. For more information on the type and target properties of the event object, see [“Event object properties” on page 337](#).

Event summary for the UIComponent class

The following table lists the events for the UIComponent class:

| Event | Description |
|----------------|--|
| focusIn | Broadcast when an object receives focus. |
| focusOut | Broadcast when an object loses focus. |
| hide | Broadcast when an object's state changes from visible to invisible. |
| invalid | Broadcast when objects should be redrawn on the screen. This is called internally by Flex or by component authors. |
| keyDown | <p>Broadcast when a key is pressed.</p> <p>The event object for this event has the following properties in addition to type and target:</p> <p>code (Number) The virtual key code of the last key pressed.</p> <p>ascii (Number) The ASCII value of the last key pressed.</p> <p>shiftKey (Boolean) True if the Shift key was pressed.</p> <p>ctrlKey (Boolean) True if the Control key was pressed.</p> |
| keyUp | <p>Broadcast when a key is released.</p> <p>The event object for this event has the following properties in addition to the type and target properties:</p> <p>code (Number) The virtual key code of the last key pressed.</p> <p>ascii (Number) The ASCII value of the last key pressed.</p> <p>shiftKey (Boolean) True if the Shift key was pressed.</p> <p>ctrlKey (Boolean) True if the Control key was pressed.</p> |
| resize | Broadcast when objects are resized. This is called internally by Flex or by component authors. |
| show | Broadcast when Flex displays objects. This is called internally by Flex or by component authors. |
| valid | Called internally by Flex or by component authors. |
| valueCommitted | Broadcast when the value of a property changes. This is called internally by Flex. |

The following example captures the ASCII values of each character and prints them to the `TextArea` when the low-level `keyDown` event is triggered. The `TextArea` control is disabled, which prevents the user from changing the values of that field.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:Script>
    <![CDATA[
      function logKey(myKey) {
        ta1.text = ta1.text + " " + myKey;
      }
    ]]>
  </mx:Script>

  <mx:TextArea id="ta1" text="" enabled="false" width="600"
    keyDown="logKey(event.code)" />
</mx:Application>
```

</mx:Application>

Event summary for the UIObject class

The UIObject class events refer to the visibility, instantiation, and size of objects in the application. Many of the events are also related to the mouse. For more information, see [“Handling mouse events” on page 352](#).

The following table describes the events for the UIObject class:

| Event | Description |
|------------------|--|
| creationComplete | Broadcast when the object has finished its construction, measuring, layout, and drawing. |
| dragBegin | Broadcast to the drag initiator when the user makes a gesture that starts a drag-and-drop operation. No Flex components respond to this event; any custom components that you create can use this event. |
| dragComplete | Broadcast to the drag initiator when the drag operation completes, either when you drop the drag data onto a drop target or when you end the drag-and-drop operation without performing a drop. You can use this event to perform any final cleanup of the drag-and-drop operation. For example, if you drag a List control item from one list to another, you can delete the List control item from the source if you no longer need it. |
| dragDrop | Broadcast to the drop target when the mouse is released over it. You use this event handler to add the drag data to the drop target. |
| dragEnter | Broadcast to the drop target when a drag initiator passes over the target. Only components that define a handler for this event can be drop targets. Within the handler, you can change the appearance of the drop target to provide visual feedback to the user that the component can accept the drag. For example, you can draw a border around the drop target, or give focus to the drop target. |
| dragExit | Broadcast to the drop target when the user drags outside the drop target, but does not drop the data onto the target. You can use this event to restore the drop target to its normal appearance if you modified its appearance as part of handling the <code>dragEnter</code> event. |
| dragOver | Broadcast to the drop target when the user moves the mouse over the target. You can handle this event if you want to perform additional logic before allowing the drop, such as dropping data to various locations within the drop target, reading keyboard input to determine if the drag-and-drop action is a move or copy of the drag data, or providing different types of visual feedback based on the type of drag- and-drop action. |
| draw | Broadcast when an object is about to draw its graphics. |
| effectEnd | Broadcast when an effect finishes. |
| effectStart | Broadcast when an effect starts. |
| initialize | Broadcast when the object is finished instantiation. |

| Event | Description |
|-----------------------------------|--|
| <code>mouseChangeSomewhere</code> | <p>Broadcast when the mouse moves over a different component, to all listeners that have registered for it, regardless of whether the mouse is over them.</p> <p>Listening to this event is more efficient than listening to the <code>mouseMoveSomewhere</code> event, which triggers whenever the mouse moves, even if it is still over the same component.</p> <p>The event object for this event has the following properties, in addition to the <code>type</code> and <code>target</code> properties:</p> <ul style="list-style-type: none"> <code>previousRelatedTarget</code> The component the mouse was over previously. <code>relatedTarget</code> The component the mouse is currently over. |
| <code>mouseDown</code> | <p>Broadcast when the cursor is over the component and the user releases the mouse button.</p> <p>The event object for this event has the following properties, in addition to the <code>type</code> and <code>target</code> properties:</p> <ul style="list-style-type: none"> <code>eventObject.target.mouseX</code> <code>eventObject.target.mouseY</code> |
| <code>mouseDownSomewhere</code> | <p>Broadcast when the mouse is clicked, to all listeners that have registered for this event, regardless of whether the mouse is over them.</p> <p>The mouse might be over a subcontrol, which is internal to a component, such as a row of a <code>DataGrid</code> control. You can walk up the parent chain looking for a component that has a <code>mouseDown</code> listener.</p> <p>The event object for this event has the following property, in addition to the <code>type</code> and <code>target</code> properties:</p> <ul style="list-style-type: none"> <code>relatedTarget</code> The component the mouse is currently over. |
| <code>mouseMove</code> | Broadcast when the mouse pointer moves. |
| <code>mouseMoveSomewhere</code> | <p>Broadcast when the mouse moves, to all listeners that have registered for it, regardless of whether the mouse is over them.</p> <p>Listening to this event is less efficient than listening to the <code>mouseChangeSomewhere</code> event.</p> <p>The event object for this event has the following property, in addition to the <code>type</code> and <code>target</code> properties:</p> <ul style="list-style-type: none"> <code>relatedTarget</code> The component the mouse is currently over. |
| <code>mouseOver</code> | Broadcast when the user hovers the mouse pointer over a control. |
| <code>mouseOut</code> | Broadcast when the user stops holding the mouse pointer over a control. |
| <code>mouseUp</code> | Broadcast when the user releases the mouse button. |
| <code>mouseUpSomewhere</code> | <p>Broadcast when the mouse button is released, to all listeners that have registered for this event, regardless of whether the mouse is over them.</p> <p>The mouse might be over a subcontrol, which is internal to a component, such as a row of a <code>DataGrid</code> control. You can walk up the parent chain looking for a component that has a <code>mouseUp</code> listener.</p> <p>The event object for this event has the following property in addition to the <code>type</code> and <code>target</code> properties:</p> <ul style="list-style-type: none"> <code>relatedTarget</code> The component the mouse is currently over. |

| Event | Description |
|--------|--|
| move | <p>Broadcast when the object has moved.</p> <p>The event object for this event has the following properties, in addition to the <code>type</code> and <code>target</code> properties:</p> <p><code>oldX</code> (Number) The original x coordinate.</p> <p><code>oldY</code> (Number) The original y coordinate.</p> |
| resize | <p>Broadcast when the subobjects are being unloaded. This is called internally by Flex or by component developers.</p> <p>The event object for this event has the following properties, in addition to the <code>type</code> and <code>target</code> properties:</p> <p><code>oldWidth</code> (Number) The original width, in pixels.</p> <p><code>oldHeight</code> (Number) The original height, in pixels.</p> |

CHAPTER 14

Creating ActionScript Components

Macromedia Flex lets you define custom components in ActionScript as part of your application. Using custom components, you can encapsulate your application logic as modules that you can reuse in a single application, or share across multiple applications.

You can also define custom ActionScript components to extend the Flex component library. For example, you can create a customized Button, Tree, or DataGrid component as an ActionScript component.

This chapter describes how to create custom components in ActionScript, and includes examples of creating components that extend the Flex component hierarchy.

Contents

| | |
|--|-----|
| About ActionScript components | 359 |
| Defining custom user-interface components | 362 |
| Passing data to a custom tag | 362 |
| Defining events in ActionScript components | 363 |
| Using the createChildren() and createClassObject() methods | 367 |
| Using metadata keywords | 368 |
| Adding ActionScript components to the Flex environment | 373 |
| Defining nonvisual components | 375 |

About ActionScript components

You create reusable components using ActionScript, and reference these components in your Flex applications as MXML tags. Components created in ActionScript can contain graphical elements, define custom business logic, or extend existing Flex components. They can inherit from any components available in Flex.

For example, you can define a custom button, derived from the Flex Button control, as the following example shows:

```
class myControls.MyButton extends mx.controls.Button {  
    function MyButton() {  
        ...  
    }  
}
```

In this example, you write your MyButton control to the MyButton.as file, and you store the file in the myControls subdirectory of the root directory of your Flex application. The fully qualified class name of your component reflects its location. In this example, the component's fully qualified class name is myControls.MyButton.

You can reference your custom Button control from a Flex application file, such as MyApp.mxml, as the following example shows:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"  
    xmlns:cmp="myControls.*" >  
    <cmp:MyButton label="Jack" />  
</mx:Application>
```

In this example, you first define the cmp namespace that defines the location of your custom component in the application's directory structure. You then reference the component as an MXML tag using the namespace prefix.

Typically, you put custom ActionScript components in directories that are in the ActionScript classpath. These include your application's root directory, the *flex_app_root*/WEB-INF/flex/user_classes directory, or any directory that you specify in the <actionscript-classpath> tag in the flex-config.xml file. For more information, see [“Editing the ActionScript classpath” on page 814](#).

If the component is at the top level of its directory structure (or not in a package), you can specify an asterisk (*) for the namespace. Adding a tag prefix is optional in this case.

If you store the MyButton.as file in the root directory of your Flex application, which is the same directory as the MyApp.mxml file, its fully qualified class name is MyButton. You must still provide a namespace, as the following example shows:

```
<mx:Application xmlns:cmp="*" xmlns:mx="http://www.macromedia.com/2003/mxml" >  
    <cmp:MyButton label="Click Me" />  
</mx:Application>
```

For more information on setting the directory location of your custom components, specifying your namespace, and determining your classpath, see [“Specifying the component namespace” on page 374](#).

Benefits of custom components

Defining your own components in ActionScript has several benefits. Components let you divide your applications into individual modules that you can develop and maintain separately. By implementing commonly used logic within custom components, you can build a suite of reusable components that you can share among multiple Flex applications.

In addition, you can base your custom components on the set of Flex components by extending from the Flex class hierarchy. You can create custom versions of Flex visual controls, as well as custom versions on nonvisual components, such as data validators, formatters, and effects.

Types of custom components

You can create the following types of components in ActionScript:

- **User-interface components** User-interface components contain both processing logic and visual elements. These components usually extend the Flex component hierarchy. You can extend from the `UIObject` and `UIComponent` classes, or any of the Flex components, such as `Button`, `ComboBox`, or `DataGrid`. Your custom ActionScript component inherits all of the public methods and properties of its parent class.
- **Nonvisual components** Nonvisual components define no visual elements. A nonvisual component is an ActionScript class that does not extend the `UIObject`, `UIComponent`, or `MovieClip` class. Using nonvisual components can provide greater efficiency at runtime.

About scope

In an ActionScript component, the scope is the component itself and not the application or other MXML file that uses the component. As a result, the `this` keyword inside the custom component refers to the component instance and not the Application object.

Nonvisual ActionScript components do not have access to their parent application with the `parentApplication` property. However, you can access the top-level Application object using the `mx.core.Application.application` property.

For more information on using these properties, see [“About the Application object” on page 329](#).

Defining getters and setters

The recommended way of exposing properties in your class file is with a pair of getters and setters. These functions must be public. The advantage of getters and setters is that you can calculate the return value in a single place and trigger events when the variable changes.

You define getter and setter functions using the `get` and `set` method properties within a class definition block.

The following example declares a getter and setter for the `myName` variable:

```
private var _myName:String = "Fred";
public function get myName():String {
    return _myName;
}
public function set myName(name:String) {
    _myName = name;
}
```

The local value of `_myName` is private, while the getter and setter methods are public.

In getters and setters, you cannot use the same name of the property in the function name. As a result, you should use some convention, such as prefixing the member name with an underscore character (`_`).

Defining custom user-interface components

You can extend Flex components to customize them by extending from the component's fully qualified class name. The fully qualified class name is not the same as the component name. For example, to extend from the Flex Grid container, you define your class as the following example shows:

```
class myControls.DeleteTextArea extends mx.containers.Grid {  
    ...  
}
```

For the list of fully qualified class names for Flex components, see *Flex ActionScript and MXML API Reference*.

Passing data to a custom tag

To make your ActionScript components reusable, you often define them to accept input properties. To add properties to your ActionScript components, you define variables within the component. As long as you do not define the variable as `private`, you can set the value of the variable using an MXML tag property.

The following example defines the variable `startMessage`:

```
class DeleteTextArea extends mx.controls.TextArea {  
    public var startMessage:String;  
    function myInit():Void {  
        this.text = startMessage;  
    }  
    function keyDown(e:Object):Void {  
        var k:Number = e.code;  
        // Delete key corresponds to a value of 46  
        if (k==46) setText("");  
    }  
}
```

You can use the `initialize` event to call an event for your custom ActionScript component. The `initialize` event triggers after the component has been instantiated. As a result, you can use it to set property values that you normally cannot set before the component exists.

The following MXML calls the `myInit()` method of the ActionScript component:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" xmlns="*" >  
    <mx:Script><![CDATA[  
        function callMyInit() {  
            myTA.myInit();  
        }  
    ]]></mx:Script>  
    <DeleteTextArea id="myTA" startMessage= "This is the start message"  
        initialize="callMyInit();" />  
</mx:Application>
```

Defining events in ActionScript components

Most Flex components inherit a large set of events from the base classes, `UIObject` and `UIComponent`. From the `UIComponent` class, components inherit events such as `focusIn`, `focusOut`, `keyDown`, and `keyUp`. From the `UIObject` class, components inherit events such as `mouseDown`, `mouseUp`, `mouseOver`, and `mouseOut`. For a complete list of `UIObject` and `UIComponent` events, see [“Using base class events” on page 353](#).

Components can emit and consume events. In most cases, you want your component to emit an event and the MXML application to consume and handle it.

Custom components that extend existing Flex classes inherit those events. For example, if you extend the `mx.controls.Button` class, you have the set of click-related events at your disposal, in addition to the events that all controls inherit, such as `mouseOver` and `mouseDown`.

You can use events in ActionScript components using the `addEventListener()` and `dispatchEvent()` methods. You must also use the `Event` metadata keyword at the top of the class file to specify any events that are being broadcast by the new component. Also, because your custom components have their own scope, you might be required to use the `Delegate` utility class to let them access alternative function or object scopes.

The following example creates a container with a child `Button` control, and registers the `Button` control's `click` event to point to the new event, `MyEvent`:

```
[Event("MyEvent")]
class MyComponent extends mx.containers.VBox {
    // Required for Delegate.
    import mx.utils.Delegate;
    function MyComponent() {
        addEventListener("initialize",myInit);
    }
    function myInit() {
        var btn;
        var okDelegate:Object = Delegate.create(this, onOk);
        btn=createChild(mx.controls.Button,"btn",{label:"Ok"},true);
        btn.addEventListener("click",okDelegate);
    }
    function onOk() {
        this.dispatchEvent({type:"MyEvent"});
    }
}
```

Handling the initialize event

When Flex finishes creating a component, it emits the component's `initialize` event. This event is used by MXML developers to populate data, debug, or perform some other function before the user starts interacting with the application.

Because nearly all classes extend the `UIObject` class, the `initialize` event is already supported in custom components. To define a handler for it, you add the `initialize` property to the component's MXML tag, and then add ActionScript code that processes the event, as the following example shows:

```
<?xml version="1.0"?>
```

```

<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" xmlns="*">
  <mx:Script><![CDATA[
    function myInit() {
      trace('init greensquare');
    }
  ]]></mx:Script>
  <MyComponent initialize="myInit();" />
</mx:Application>

```

Handling mouse events

All visual components that inherit from the `UIObject` class support a number of navigational mouse events, including the following:

- `mouseover`
- `mouseout`
- `mousedown`
- `mouseup`

These events do not include the `click` event. For a complete list of events supported by visual components, see the information about the control in the control's chapter.

To use the common mouse events, you point to a handler in your MXML file. You do not need to add any additional code to the component source code.

The following example changes the `alpha` (transparency) of the custom component when the mouse moves over the component, and again when the mouse moves away from the component:

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" xmlns="*">
  <mx:Script><![CDATA[
    var startAlpha:Number = 100;
    function myInit() {
      myGS.alpha=startAlpha;
    }
    function changeAlpha(curAlpha:Number) {
      myGS.alpha=curAlpha;
    }
  ]]></mx:Script>
  <MyComponent id="myGS" mouseOver="changeAlpha(50);"
    mouseOut="changeAlpha(startAlpha);" />
</mx:Application>

```

To handle an event that is not supported by the current parent class, such as a `click` event on a `UIObject`, you must edit the component class file. However, to add a `click` event to your component, it is sometimes easier to extend the `Button` or `SimpleButton` class than it is to write the code to support a `click` event.

For information on defining new events and event handlers for your custom component, see [“Emitting events”](#) next.

Emitting events

You can define an event that is not inherited from the component's parent class, such as a click for a control that is not a subclass of the Button class. In the following example, Flex throws an error because the custom component does not emit a `click` event:

```
<MyComponent id="myGS" click="changeAlpha(0);" />
```

If you try to use a click handler in the MXML file, you get an error similar to the following:

```
Error: unknown attribute 'click' on MyComponent
```

This means that you have to go back to the component's `ActionScript` class and tell the component to emit a `click` event. You do this by adding a call to `dispatchEvent()` in the component's `ActionScript` class file. You must also include the `Event` metadata keyword so that Flex recognizes the dispatched event. For more information on the `dispatchEvent()` method, see [“Handling events” on page 337](#).

The following example adds a metadata keyword identifying `click` as an event that this component can emit, and then dispatches the `click` event when the `onRelease()` method is triggered. In this case, the `click` event causes a change in the instance's `alpha` property.

```
[Event("click")]
class MyComponent extends mx.core.UIObject {
    function MyComponent() {
    }
    function init() {
        var w:Number = 200;
        var h:Number = 200;
        //draw the box
        lineStyle(1, 0x666600, 100);
        beginFill(boxColor, 100);
        moveTo(0, 0);
        lineTo(w-1, 0);
        lineTo(w-1, h-1);
        lineTo(0, h-1);
        lineTo(0, 0);
        endFill();
        super.init();
    }
    function onRelease():Void {
        dispatchEvent({ type: "click" });
    }
}
```

The following MXML file handles the `click` event within an `<mx:Script>` block:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" xmlns="*" >
    <mx:Script><![CDATA[
        var startAlpha:Number = 100;
        var curState:Number = 50;
        function myInit() {
            gs.alpha=startAlpha;
        }
        function modAlpha(curAlpha:Number) {
```

```

        gs.alpha=curAlpha;
        if (curState==100) {
            curState=50;
        } else {
            curState=100;
        }
    }
]]></mx:Script>
<MyComponent id="gs" initialize="myInit();" click="modAlpha(curState);" />
</mx:Application>

```

For an example of a custom component that emits and handles its events, see [“Creating compound components” on page 366](#).

Creating compound components

Compound components are components that include the assets of multiple controls inside them. They might be graphical assets or a combination of graphical assets and classes. For example, you can create a component that includes a button and a text field, or a component that includes a button, a text field, and a validator.

When you create compound components, you should instantiate the controls inside the component’s class file. Assuming that some of these controls have graphical assets, you must plan the layout of the controls that you are including, and set properties such as default values in your class file. You must also ensure that you import all the necessary classes that the compound component uses.

Because the class extends one of the base classes, such as `mx.core.UIComponent`, and not a controls class like `mx.controls.Button`, you must instantiate each of the controls as children of the custom component and arrange them on the screen.

Properties of the individual controls are not accessible from the MXML author’s environment unless you design your class to allow this. For example, if you create a new component that extends the `UIComponent` class and uses a `Button` and a `TextArea` component, you cannot set the label text in the MXML tag because you do not directly extend the `Button` class.

To instantiate controls inside your compound component, use the `createClassObject()` method inside the `createChildren()` method. For more information, see [“Using the `createChildren\(\)` and `createClassObject\(\)` methods” on page 367](#).

This section uses an example component, called `CompoundComponent`, that combines a `Button` control and a `TextArea` control. It handles the `click` event of the `Button` control and writes a message to the `TextArea` control.

This component handles events (and doesn’t just emit them), so it includes an event listener and an event handler in the class file.

The `layoutChildren()` method handles the layout of the controls inside the component. This method calls the control’s `move()` method to arrange the `Button` control centered below the `TextArea` control.

The following ActionScript class file creates a component that instantiates TextArea and Button controls:

```
// Import all necessary classes.
import mx.core.UIComponent;
import mx.controls.Button;
import mx.controls.TextArea;

[Event("click")]
class CompoundComponent extends UIComponent {
    function CompoundComponent() {
    }
    function init() {
        super.init();
    }
    // Declare two children member variables.
    var text_mc:TextArea;
    var mode_mc:Button;
    function createChildren():Void {
        if (mode_mc == undefined)
            createClassObject(Button, "mode_mc", 1, { });
        if (text_mc == undefined)
            createClassObject(TextArea, "text_mc", 0, { preferredWidth: 150,
            editable: false });
        mode_mc.addEventListener("click", this);
        mode_mc.label = "Click Me";
    }
    function layoutChildren():Void {
        mode_mc.move(text_mc.width/2-5, 50);
    }
    // Handle events that are dispatched by the children.
    function handleEvent(evt:Object):Void {
        if (evt.type == "click")
            text_mc.text = "the button was clicked";
    }
}
```

Using the createChildren() and createClassObject() methods

Components implement the createChildren() method to create subobjects (such as other components) in the component. Rather than calling the subobject's constructor in the createChildren() method, call the createClassObject() method to instantiate a subobject of your component.

The createClassObject() method has the following signature:

```
createClassObject(className, instanceName, depth, initObject)
```

The following table describes the arguments:

| Argument | Type | Description |
|--------------|--------|---------------------------|
| className | Object | The name of the class. |
| instanceName | String | The name of the instance. |

| Argument | Type | Description |
|------------|--------|---|
| depth | Number | The depth for the instance. |
| initObject | Object | The object that contains the initialization properties. |

To call the `createClassObject()` method, you must know what those children are (for example, a border or a button that you always need), because you must specify the name and type of the object, plus any initialization arguments in the call to `createClassObject()`.

The following example calls the `createClassObject()` method to create a new `Label` object for use inside a component:

```
up_mc.createClassObject(Label, "label_mc", 1); // Create a label in the holder
```

You set properties in the call to the `createClassObject()` method by adding them as part of the `initObject` argument. The following example sets the value of the `label` property:

```
form.createClassObject(CheckBox, "cb", 0, {label:"Check this"});
```

The following example creates `TextInput` and `SimpleButton` components:

```
function createChildren():Void {
    if (text_mc == undefined)
        createClassObject(TextInput, "text_mc", 0, { preferredWidth: 80,
            editable:false });
    text_mc.addEventListener("change", this);
    text_mc.addEventListener("focusOut", this);
    if (mode_mc == undefined)
        createClassObject(SimpleButton, "mode_mc", 1, { falseUpSkin:
            modeUpSkinName, falseOverSkin: modeOverSkinName, falseDownSkin:
            modeDownSkinName });
    mode_mc.addEventListener("click", this);
}
```

If your component is a container, and you do not know exactly which children it contains, call the `createComponents()` method. This method creates all child objects. For more information on using the `createComponents()` method, see [“Using the createComponent\(\) method” on page 582](#).

Using metadata keywords

The Flex compiler recognizes component metadata statements in your ActionScript class files. The metadata tags define component attributes, data binding properties, events, and other properties of the component. Flex interprets these statements during compilation; they are never interpreted during runtime.

Working with metadata keywords

Metadata statements are associated with a class declaration or an individual data field. They are bound to the next line in the ActionScript file. When defining a component property, add the metadata tag on the line before the property declaration. When defining component events or other aspects of a component that affect more than a single property, add the metadata tag outside the class definition so that the metadata is bound to the entire class.

In the following example, the `Inspectable` metadata keywords apply to the `flavorStr`, `colorStr`, and `shapeStr` properties:

```
[Inspectable(defaultValue="strawberry")]
public var flavorStr:String;

[Inspectable(defaultValue="blue")]
public var colorStr:String;

[Inspectable(defaultValue="circular")]
public var shapeStr:String;
```

Metadata tags

The following table describes the metadata tags that you can use in ActionScript class files:

| Tag | Description |
|---------------------------------------|---|
| <code>ChangeEvent</code> | Identifies events that cause data binding to occur. For more information, see “ChangeEvent” on page 369 . |
| <code>Effect</code> | Defines the valid property name for the tag’s effect. For more information, see “Effect” on page 370 . |
| <code>Embed</code> | Imports JPEG, GIF, PNG, SVG, and SWF files at compile time. Also imports image assets from SWC files. This is functionally equivalent to the MXML <code>@Embed</code> syntax, as described in Chapter 9, “Importing Images” . For more information, see “Importing images using Embed in ActionScript” on page 293 . |
| <code>Event</code> | Describes the events that the component emits. For more information, see “Event” on page 370 . |
| <code>IconFile</code> | Identifies the filename for the icon that represents the component in the Insert Bar of Flex Builder. |
| <code>Inspectable</code> | Defines an attribute exposed to component users in the attribute hints and Tag inspector of Flex Builder. Also limits allowable values of the property. For more information, see “Inspectable” on page 371 . |
| <code>NonCommittingChangeEvent</code> | Identifies an event as an interim trigger. For more information, see “NonCommittingChangeEvent” on page 372 . |
| <code>Style</code> | Describes a style property allowed on the component. For more information on using the <code>Style</code> metadata keyword, see “Style” on page 373 . |

The following sections describe the component metadata tags in more detail.

ChangeEvent

Use the `ChangeEvent` metadata keyword to generate one or more component events when changes are made to properties.

The `ChangeEvent` metadata keyword has the following syntax:

```
[ChangeEvent("event_name"[,...])]  
property_declaration or get/set function
```

You can use this keyword only with variable declarations or getter or setter functions, although it is not required.

In the following example, the component generates the `change` event when the value of the `flavorStr` property changes:

```
[ChangeEvent("change")]  
public var flavorStr:String;
```

When the event specified in the metadata occurs, Flash informs whatever is bound to the property that the property has changed.

You can also instruct your component to generate an event when a getter or setter function is called, as the following example shows:

```
[ChangeEvent("change")]  
function get selectedDate():Date
```

In most cases, you set the `change` event on the getter function, and dispatch the event on the setter function.

You can register multiple `change` events in the metadata so that more than one event is generated when the property changes, as the following example shows:

```
[ChangeEvent("change1")]  
[ChangeEvent("change2")]  
[ChangeEvent("change3")]
```

Any one of those events indicates a change to the variable. They do not all have to occur to indicate a change.

Effect

The `Effect` metadata keyword defines the name of the property that you can assign to an effect for the MXML tag.

The `Effect` metadata keyword has the following syntax:

```
[Effect("effect_name")]
```

The *effect_name* generally matches the *event_name* listed in the `Effect` metadata plus the word *Effect*. The following example means that a tag may have a `resizeEffect` property assigned to the effect to play when the `resize` event triggers:

```
[Effect("resizeEffect")]
```

Event

Use the `Event` metadata keyword to define events dispatched by this component. Add the `Event` statements outside the class definition in the `ActionScript` file so that they are bound to the class and not a particular member of the class.

The `Event` metadata keyword has the following syntax:

```
[Event("event_name")]
```

The following example identifies the `myClickEvent` event as an event that the component can dispatch:

```
[Event("myClickEvent")]
```

If you do not identify an event in the class file with the `Event` metadata keyword, the compiler ignores the event during compilation, and Flex ignores this event triggered by the component during runtime. The metadata for events is inherited from the parent class, however, so you do not need to tag events that are already tagged with the `Event` metadata keyword in the parent class.

The following example shows the `Event` metadata for the `UIObject` class, which handles the `resize`, `move`, and `draw` events:

```
[Event("resize")]
[Event("move")]
[Event("draw")]
class mx.core.UIObject extends MovieClip {
    ...
}
```

Inspectable

You specify the user-editable (or *inspectable*) parameters of a component in the class definition for the component using the `Inspectable` metadata keyword. If you tag a property as inspectable, it appears in the Tag Inspector or attribute hints of Flex Builder.

The `Inspectable` metadata statement must immediately precede the property's variable declaration to be bound to that property.

The `Inspectable` metadata keyword has the following syntax:

```
[Inspectable(value_type=value[,attribute=value,...])]
property_declaration name:type;
```

The following table describes the properties of the `Inspectable` metadata keyword:

| Property | Type | Description |
|---------------------------|------------------|---|
| <code>category</code> | String | (Optional) Groups the property into a specific subcategory in the Property inspector of the Flash user interface. |
| <code>defaultValue</code> | String or Number | (Required) Sets a default value for the inspectable property. This property is required if used in a getter or setter function. The default value is determined from the property definition. |
| <code>enumeration</code> | String | (Optional) Specifies a comma-delimited list of legal values for the property. Only these values are allowed; for example, <code>item1</code> , <code>item2</code> , <code>item3</code> . |
| <code>environment</code> | String | (Optional) Notes which inspectable properties should not be allowed (<code>none</code>), which are used only for Flash (<code>Flash</code>), and which are used only by Flex and not Flash (<code>MXML</code>). |

| Property | Type | Description |
|-------------------------|--------|--|
| <code>format</code> | String | (Optional) Indicates that the property holds a value with a file path. |
| <code>listOffset</code> | Number | (Optional) Added for backward compatibility with Flash MX components. Used as the default index into a List value. |
| <code>name</code> | String | (Optional) A display name for the property; for example, <code>Font Width</code> . If not specified, use the property's name, such as <code>_fontWidth</code> . |
| <code>type</code> | String | (Optional) A type specifier. If omitted, use the property's type. The following values are acceptable: <ul style="list-style-type: none"> • Array • Object • List • String • Number • Boolean • Font Name • Color If the property is an array, you must list the valid values for the array. |
| <code>variable</code> | String | (Optional) Added for backward compatibility with Flash MX components. Used to specify the variable to which this parameter is bound. |
| <code>verbose</code> | Number | (Optional) Indicates that this inspectable property should be displayed in the Flash user interface only when the user indicates that verbose properties should be included. If this property is not specified, Flash assumes that the property should be displayed. |

The following example defines the `enabled` parameter as inspectable:

```
[Inspectable(defaultValue=true, verbose=1, category="Other")]
var enabled:Boolean;
```

The `Inspectable` keyword supports loosely typed properties, as the following example shows:

```
[Inspectable("danger", 1, true, maybe)]
```

NonCommittingChangeEvent

The `NonCommittingChangeEvent` metadata keyword identifies an event as an interim trigger. You use this keyword for properties that might change often, but for which you do not want validation to occur on every change.

An example of this is if you tied a validator function to the text property of a `TextInput` control. The text property changes on every keystroke, but you do not want to validate the property until the user presses the Enter key or changes focus away from the field. The `NonCommittingChangeEvent` keyword lets you trigger validation when the user is done editing the text field.

The `NonCommittingChangeEvent` metadata keyword has the following syntax:

```
[NonCommittingChangeEvent("event_name")]
```

In the following example, the component is aware that the property has changed if the change is triggered; however, that change is not final. There is another `ChangeEvent` keyword that probably triggers when the change is final:

```
[Event("change")]
[ChangeEvent("valueCommitted")]
[NonCommittingChangeEvent("change")]
function get text():String {
    return getText();
}
function set text(t):Void {
    setText(t);
}
```

Style

The `Style` metadata keyword describes a style property allowed for the component. The `Style` metadata keyword has the following syntax:

```
[Style(name=style_name[,attribute=value,...])]
```

The following table describes the properties for the `Style` metadata keyword:

| Option | Type | Description |
|------------|--------|---|
| name | String | (Required) The name of the style. |
| type | String | The type of data. |
| format | String | Units of the property. |
| inheriting | String | Whether the property is inheriting. Valid values are <code>yes</code> and <code>no</code> . |

The following example shows the `textSelectedColor` property:

```
[Style(name="textSelectedColor",type="Number",format="Color",inherit="yes")]
```

Adding ActionScript components to the Flex environment

You can use the ActionScript component in Flex after you add it to a location that is in the Flex ActionScript classpath and specify the correct namespace for that component. This section describes how to do this.

You can also compile the ActionScript file into a SWC file and deploy the component as a SWC file. For more information, see [“Using SWC files” on page 800](#).

Determining the ActionScript classpath

Macromedia recommends that you put components shared by multiple applications in the system's ActionScript classpath, and put application-specific components in subdirectories of the application's root directory. Flex uses XML namespaces to locate components in your application's directory structure.

The following rules can help you organize your custom components:

1. MainApp.mxml can reference components in /dir1/dir2, and its subdirectories.
2. ActionScript components in /dir1/dir2 must have fully qualified class names defined relative to the location of the application's root directory. For example, if you define a custom component in the file /dir1/dir2/msg/controls/PieChart.as, its fully qualified class name must be `msg.controls.PieChart`, assuming that the application is in /dir1/dir2/.
3. ActionScript components can reference components located in the classpath.
The component search order in the classpath is based on the order of directories listed in the classpath. The search is based on fully qualified class names. For example, if you have the file `myButton.as` under /dir1/dir2, and under `WEB-INF/flex/user_classes`, Flex uses the file under /dir1/dir2.
4. The `<mx:Script>` tag in the MainApp.mxml file, and in dependent MXML components, can reference components located in the ActionScript classpath.

Specifying the component namespace

Depending on where a component is located, you specify its namespace in one of the following ways:

- If component files are in the same directory as the application file or in the ActionScript classpath directory (not a subdirectory) that is defined in the `flex-config.xml` file, you can refer to them as the following example shows. In this example, the local namespace (*) is mapped to the prefix `local`.

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
  xmlns:local="*" >
  <local:MyButton />
  ...
</mx:Application>
```

If the same file exists in the ActionScript classpath directory and the application directory, Flex uses the file in the application file directory.

- If the component is in a subdirectory of the directory that contains the MXML application file, you use a namespace that specifies the subdirectory. The following code declares a component that is in the `custom/components` subdirectory of the directory that contains the MXML file in which it is used:

```
<m1:MyComponent xmlns:m1="custom.components.*" />
```

If the same file exists in the ActionScript classpath directory and the application directory, Flex uses the file in the application file directory.

- If the component is in a subdirectory of the ActionScript classpath directory that is defined in the `flex-config.xml` file, you use a namespace that specifies the subdirectory. The following code declares an MXML component that is in the `flex_app_root/WEB-INF/flex/user_classes/global/custom/mxml` directory:

```
<m1:mycomponent xmlns:m1="global.custom.mxml.*" />
```

If the same file exists in the ActionScript classpath directory and the application directory, Flex uses the file in the application file directory.

For additional information about component namespaces, see [“Defining component namespaces” on page 841](#).

Defining nonvisual components

Nonvisual components are a special kind of ActionScript component. Nonvisual components extend nonvisual Flex classes or objects, such as data validators, data formatters, and effects. To create these component types, you derive from `mx.validators.Validator`, `mx.formatters.Formatter`, or `mx.effects.TweenEffect`.

Nonvisual components can also implement the `MXMLObject` interface so that the initialization of the nonvisual component is timed properly.

The benefit of nonvisual components is that they are reusable, but they do not require the overhead of the Flex component architecture.

You are not required to extend an existing class in the Flex hierarchy to create a nonvisual component. You can define nonvisual components to implement any programming logic that does not require a user interface. In this case, you create a stand-alone class definition.

Because nonvisual components do not have visual elements, you cannot use them in your application where you use visual components, such as a Button control. You must declare nonvisual ActionScript components in the MXML file as either a child of the root tag, or as the value of a property, if the property is an object. Nonvisual ActionScript components are not allowed as children of a container, except when that container is the tag at the root of the document.

All nonvisual components must have a no-argument constructor and must follow the same rules as a user-interface component for having public variables. The following example shows a simple nonvisual component:

```
class myFacelessComp {  
    function myFacelessComp() {  
    }  
}
```

When instantiating a nonvisual component, if an argument is an object, you can only pass in a binding expression, you cannot instantiate an anonymous object inline.

Flex effect example

One example of a nonvisual component is a Flex effect. Effects extend `mx.effects.TweenEffect`, or a child of `mx.effects.TweenEffect`, and effects' constructors take an argument. For example, you can extend the `WipeRight` effect, as the following example shows:

```
class myEffects.myShowEffect extends mx.effects.WipeRight {  
    function myShowEffect(targetObj:Object) {  
        target=targetObj;  
        trace("myShowEffect constructor");  
    }  
}
```

This example is a simple modification that writes a trace statement to the error log when the effect is invoked. Otherwise, it acts like its parent class.

You can use your nonvisual component to define an effect within the `<mx:Effect>` tag. Flex requires that you always specify a `name` property for your nonvisual components when you use them in an application, as the following example shows:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
  xmlns:eff="myEffects.*">
  <mx:Effect>
    <eff:myShowEffect name="effect1" />
  </mx:Effect>
  <mx:TextInput id="text0" width="200" text="click button for effects"
    showEffect="effect1" />
  <mx:Button label="button0" click="text0.visible=!text0.visible;" />
</mx:Application>
```

Nonvisual components do not have a standard `initialize` event that other components support. Flex provides an interface that specifies a single method that is called after all of the properties have been set on a newly instantiated component. The interface has a single method, `initialized()`. Flex calls the `initialized()` method after the implementing object has been created and all properties specified on the tag have been assigned.

The `initialized()` method has the following signature:

```
initialized (Object document, String id):Void
```

The following table describes the properties of the `initialized()` method:

| Property | Type | Description |
|----------|--------|--|
| document | Object | The MXML document that created this object. |
| id | String | The ID used by the document to refer to this object. |

The following example implements the `mx.core.MXMLObject` interface, and then overrides the `initialized()` method:

```
class TempConverter implements mx.core.MXMLObject {
  public var view;
  function initialized(doc : Object, id : String) {
    view.myButton.addEventListener("click", this);
  }
  function click(event) {
    view.celsius.text=(view.fahrenheit.text-32)/1.8;
  }
}
```

For more information on creating custom formatter components in ActionScript, see [Chapter 32, “Formatting Data,” on page 725](#). For information on creating custom validator components in ActionScript, see [Chapter 28, “Managing Data in Flex,” on page 629](#).

Custom event dispatcher example

The following nonvisual component defines a generic event dispatcher. This class abstracts the event listener code from the MXML document, but still lets the MXML file define the handlers for events that this class dispatches:

```
import mx.events.*;
[Event("result")]
class MyDispatcher {
    private static function staticConstructor():Boolean {
        EventDispatcher.initialize(MyDispatcher.prototype);
        return true;
    }

    // Load an EventDispatcher
    private static var EventDispatcherDependency = EventDispatcher;

    private static var staticConstructed:Boolean = staticConstructor();
    public var addEventListener:Function;
    public var removeEventListener:Function;
    private var dispatchEvent:Function;

    function MyDispatcher() {
        // Empty constructor.
    }

    public function load():Void {
        dispatchEvent({type:"result"});
    }
}
```

The following MXML file uses the `MyDispatcher` function as a custom event dispatcher. It does not add an event listener; it triggers the custom event dispatcher when the user clicks the button. In addition, the MXML tag specifies the event handler for the result event, as the following code shows:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    xmlns:myClasses="*">
    <myClasses:MyDispatcher id="myInstance" result="statusLabel.text='Loaded';"
    />
    <mx:Button label="Load" click="myInstance.load()"/>
    <mx:Label id="statusLabel" text="" />
</mx:Application>
```

To handle other events in the custom dispatcher, you insert `Event` metadata tags at the top of the class file and add additional `dispatchEvent()` methods.

CHAPTER 15

Customizing Data Provider Controls

This chapter provides information about customizing the appearance and behavior of cells in List-based data provider controls used in Macromedia Flex applications, including List, HorizontalList, TileList, DataGrid, and Tree controls.

Contents

| | |
|--|-----|
| Creating a cell renderer | 379 |
| Creating a DataGrid header renderer | 384 |
| Displaying images in a List-based control. | 386 |
| Validating data in a List-based control | 389 |
| Formatting data in a List-based control | 391 |

Creating a cell renderer

The cell renderer application programming interface (API) is a set of properties and methods that List-based controls use to manipulate and display custom cell content for each of their rows. This customized cell can contain a prebuilt component, such as a CheckBox control, or any user interface component that you create.

You use the CellRenderer API to write a custom cell renderer as an MXML component or ActionScript class for List-based controls. Before using the CellRenderer API, you should become familiar with the List class and List-based controls. A List-based control is composed of rows that display rollover and selection highlights, are used as hit states for row selection, and provide scrolling functionality. Each row contains one cell, with the exception of the DataGrid rows, which contain multiple cells. Standard List-based controls include the List, DataGrid, and Tree controls.

Using List-based controls

By default, List cells are TextField objects that implement the CellRenderer API. You can create a custom cell renderer class or MXML component to use a different type of component as the cell for each row.

A List-based control only lays out as many rows as it can display at once; items beyond the value of the `rowCount` property do not get rows. When the list scrolls, the component moves all the rows up or down.

It is the responsibility of the cell renderer object to know how to reset its state when it is set to a new value. For example, if your cell renderer object creates an icon to display one item, it might need to remove that icon when another item is rendered with it. Assume that the cell renderer object is a container that will be filled with numerous item values over time; it has to know how to completely change itself from displaying one value to displaying another. The cell should know how to properly render undefined items, which might mean removing all old content in the cell.

About the CellRenderer API

A custom cell renderer class should extend the `UIComponent` class or any subclass of the `UIComponent` class. The following table describes `CellRenderer` methods. The only method that a cell renderer written as an MXML component must implement is the `setValue()` method. A cell renderer class must implement the `createChildren()` and `setValue()` methods.

| Method | Description |
|--|---|
| <code>CellRenderer.createChildren()</code> | <p>Cell renderers implement this method to create the subobjects in the component.</p> <p>The following example creates a <code>CheckBox</code> control:</p> <pre>function createChildren() : Void { check = createClassObject(CheckBox, "check", 1, {styleName:this, owner:this}); check.addEventListener("click", this); size(); }</pre> |
| <code>CellRenderer.getPreferredHeight()</code> | <p>Returns the preferred height of a cell.</p> <p>This method is especially important for getting the right height of text within the cell. If you set this value higher than the <code>rowHeight</code> property of the component, cells will extend above and below the rows.</p> <p>Declare the method in your own class as the following example shows. This example returns the value 16, which indicates that the cell's preferred height is 16 pixels.</p> <pre>function getPreferredHeight() : Number { return 16; }</pre> |
| <code>CellRenderer.getPreferredWidth()</code> | <p>Returns the preferred width of a cell. This method is only required when using the <code>Menu</code> component, but it is good practice to create a method stub. If you specify more width than the component has, the cell can be clipped.</p> <p>Declare the method in your own class as the following example shows. This example returns the value 3, which indicates that the cell's preferred width is three times as wide as the length of the string it is rendering.</p> <pre>function getPreferredWidth() : Number { return myString.length*3; }</pre> |

| Method | Description |
|---|---|
| <code>CellRenderer.layoutChildren()</code> | <p>Lays out the children of a component. Declare a <code>layoutChildren()</code> method in your own class to call the <code>setSize()</code> method of child objects, as the following example shows:</p> <pre>function layoutChildren() : Void { check.setSize(20, layoutHeight); }</pre> |
| <code>CellRenderer.setValue(str:String, item:Object, sel:String)</code> | <p>Sets the content to be displayed in the cell. Takes the values given and creates a representation of them within the cell. This clears up any difference in what was displayed in the cell and what needs to be displayed in the cell for the new item. Any cell could display many values during its time in the list. This is the most important method in any cell renderer. This method has the following arguments:</p> <p><code>str</code> Value to be used for the cell renderer's text, if any is needed.</p> <p><code>item</code> An object that is the entire item to be rendered. The cell renderer can use any properties of this object for rendering.</p> <p><code>sel</code> Can have the following String values:</p> <ul style="list-style-type: none"> ▪ <code>selected</code> Indicates that the row is selected . ▪ <code>highlighted</code> Indicates that the row is being rolled over . ▪ <code>normal</code> Indicates that the row is not selected or highlighted . <p>Declare the method in your own class, as the following example shows:</p> <pre>function setValue(str:String, item:Object, sel:String) : Void { check._visible = (item!=undefined); check.selected = item[getDataLabel()]; }</pre> |

If a cell does not draw quickly enough, in the `setValue()` method, you can call the `redraw()` method of the object that is being rendered. For example, if you have a label in the cell renderer, and you set `myLabel.text = itemObj.value` in the `setValue()` method, you can then call the `myLabel.redraw()` method. This updates the label and avoids a delay in refreshing the cell.

Note: Only call the `redraw()` method when there is a significant delay in refreshing the cell. Using this method can affect performance.

It is useful to give the cell renderer class a reference to the List-based control that contains it to call the `selectedIndex` property of the List-based control. To set the `selectedIndex` property to the correct value, the cell must reference the index of the item that it is currently rendering. To do so, the cell can use the property and methods described in the following table. You do not need to implement this property or these methods, but you must declare variables for them if you use them in your class.

| Property or Method | Description |
|--|---|
| <code>CellRenderer.listOwner</code> | A reference to the List component that contains the cell. Declare it in your class, as the following example shows: <code>var listOwner : UIObject</code> |
| <code>CellRenderer.getDataLabel()</code> | Returns a string that contains the name of the CellRenderer class's data field. Declare it in your class, as the following example shows: <code>var getDataLabel : Function;</code> |
| <code>CellRenderer.getCellIndex()</code> | Returns an object with two fields, <code>columnIndex</code> and <code>itemIndex</code> , that indicate the position of the cell. Declare it in your class, as the following example shows: <code>var getCellIndex : Function;</code> |

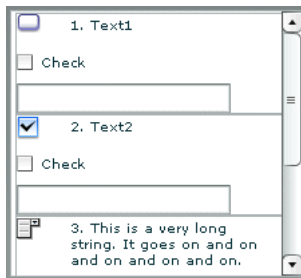
Cell renderer examples

The simplest way to use a custom cell renderer in a List-based control is to specify an MXML component as the value of the `cellRenderer` property. When you use an MXML component as a cell renderer, it can contain multiple levels of containers and controls. The MXML component must implement the `setValue()` method.

In the following example, the `cellRenderer` property of the `<mx:List>` tag is set to the `FancyCellRenderer` component. The `variableRowHeight` property is set to `true` because the MXML component exceeds the default row height.

```
<mx:List id="myList1" dataProvider="{myDP}" width="220" height="200"
  cellRenderer="FancyCellRenderer" variableRowHeight="true"/>
```

This example creates the following List control:



The following example shows the MXML code for the MXML component named `FancyCellRenderer`:

```
<mx:VBox xmlns:mx="http://www.macromedia.com/2003/mxml" width="198"
  backgroundAlpha="0" >
  <mx:Script>
```

```

var imageData:String;
var labelData:String;

var imageTable = [ "image1", "image2", "image3", "image4", "image5",
"image6", "image7", "image8", "image9" ];

[Embed(source="Accordion.png")]
var image1:String;
[Embed(source="Button.png")]
var image2:String;
[Embed(source="CheckBox.png")]
var image3:String;
[Embed(source="ComboBox.png")]
var image4:String;
[Embed(source="DataGrid.png")]
var image5:String;
[Embed(source="DateChooser.png")]
var image6:String;
[Embed(source="DateField.png")]
var image7:String;
[Embed(source="HScrollBar.png")]
var image8:String;
[Embed(source="Menu.png")]
var image9:String;

function setValue(str:String, item:Object) {
    if (item == undefined)
    {
        visible = false;
        return;
    }
    visible = true;
    var c = item.charAt(0);
    myImage.source = this[imageTable[c]];
    var s = item;
    labelData = s;
}
</mx:Script>
<mx:HBox height="100%">
    <mx:Image id="myImage" width="30" />
    <mx:Text text="{labelData}" width="150" height="100%" />
</mx:HBox>
<mx:CheckBox label="Check" />
<mx:TextInput />
</mx:VBox>

```

The following example shows a cell renderer class that displays a standard `CheckBox` control in the cells of a `List`-based control. In most cases, it is easier to use an `MXML` component instead of an `ActionScript` class as a custom cell renderer.

```

import mx.core.UIComponent
import mx.controls.CheckBox

class CheckCellRenderer extends UIComponent
{

```

```

var check : MovieClip;
var listOwner : MovieClip; // The reference we receive to the list.
var getCellIndex : Function; // The function we receive from the list.
var getDataLabel : Function; // The function we receive from the list.

function CheckCellRenderer() {
}

function createChildren() : Void {
    check = createClassObject(CheckBox, "check", 1, {styleName:this,
        owner:this});
    check.addEventListener("click", this);
    size();
}

function layoutChildren() : Void {
    check.setSize(20, layoutHeight);
    check._x = (layoutWidth-20)/2;
    check._y = (layoutHeight-16)/2;
}

function setValue(str:String, item:Object, sel:Boolean) : Void {
    check._visible = (item!=undefined);
    check.selected = item[getDataLabel()];
}

function getPreferredHeight() : Number {
    return 16;
}

function getPreferredWidth() : Number {
    return 20;
}

function click() {
    listOwner.editField(getCellIndex().itemIndex,
        getDataLabel(), check.selected);
}
}

```

Creating a DataGrid header renderer

There are situations in which you might want to change the appearance or behavior of DataGrid headers. For example, you might want to add custom ToolTips to DataGrid header cells. You can use a custom header renderer in the header cells of a DataGrid control. For more information about DataGrids, see [“DataGrid control” on page 127](#).

The following figure shows an application that contains a DataGrid control in which a header renderer displays a tooltip when the user moves the mouse pointer over the first column header:

| Name | Phone | Email |
|---------------------|--------------|---------------------------|
| Christina Coenraets | 555-219-2270 | ccoenraets@fictitious.com |
| Louis Freleigh | 555-219-2100 | lfreleigh@fictitious.com |
| Ronnie Hodgman | 555-219-2030 | rhodgman@fictitious.com |
| Joanne Wall | 555-219-2012 | jwall@fictitious.com |
| Maurice Smith | 2192012 | maurice@fictitious.com |
| Mary Jones | 555-219-2000 | mjones@fictitious.com |
| | | |
| | | |

MXML code

The following example shows the MXML code for the application. Data in the `<mx:Model>` object is bound to the DataGrid control's `dataProvider` property to populate the DataGrid control. The `headerRenderer` property of the first `DataGridColumn` contains a binding expression that sets its value to a custom header renderer named `myRenderer`, which is implemented as an ActionScript class in a file named `myRenderer.as`.

```
<?xml version="1.0" ?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">

    <mx:Model id="employeeModel" source="employees.xml"/>

    <mx:DataGrid id="dg" width="500" height="200"
        dataProvider="{employeeModel.employee}">
        <mx:columns>
            <mx:Array>
                <mx>DataGridColumn columnName="name" headerText="Name"
                    headerRenderer="{myRenderer}"/>
                <mx>DataGridColumn columnName="phone" headerText="Phone"/>
                <mx>DataGridColumn columnName="email" headerText="Email"/>
            </mx:Array>
        </mx:columns>
    </mx>DataGrid>
</mx:Application>
```

The following example is part of a file named `employees.xml` that contains the data source for the

`<mx:Model>` tag:

```
<?xml version="1.0" ?>
<employees>
  <employee>
    <name>Christina Coenraets</name>
    <phone>555-219-2270</phone>
    <email>ccoenraets@fictitious.com</email>
    <active>true</active>
  </employee>
  <employee>
    <name>Louis Freleigh</name>
    <phone>555-219-2100</phone>
```

```

        <email>lfreligh@fictitious.com</email>
        <active>true</active>
    </employee>
...
</employees>

```

Header renderer code

The following example shows the source code for the header renderer class, which is in a file named `myRenderer.as`. The bold text indicates where the tooltip for the first column header is created in the `createChildren()` method.

```

import mx.controls.Label;
class myRenderer extends mx.controls.Label {
    var lbl : MovieClip;
    var listOwner : MovieClip; //Reference to the parent DataGrid.

    function myRenderer(){
    }
    public function createChildren():Void {
        lbl = createClassObject(Label, 'label', 1, {toolTip: 'User names here.'});
        lbl.text = "Name";
    }

    public function getPreferredHeight():Number {
        return listOwner.rowHeight;
    }

    public function layoutChildren():Void {
        lbl.setSize(80, listOwner.rowHeight);
    }









    function setValue(str:String, item:Object, sel:Boolean) : Void {
    }
}

```

Displaying images in a List-based control

Sometimes images convey information better than words, and there might be times when you want to use images in the cells of a List-based control. Incorporating images into a custom cell renderer requires very little code beyond what is needed for a basic cell renderer. In a cell renderer class, you use `[Embed...]` metadata tags to embed images into variables at compile-time. You then display each image by creating an `mx.controls.Image` object that uses an embedded image as the value of its `source` property.

The following figure shows a DataGrid control that displays text in one column and images in another column. The first column uses a standard cell renderer object. The second column uses a custom cell renderer object to display images.

| name | direction |
|------------|---|
| California |  |
| New York |  |
| Alberta |  |
| Texas |  |
| Canada |  |
| Oakland |  |
| Kansas |  |
| Detroit |  |

MXML code

The following example shows the MXML code for the application. Data in the `<mx:Model>` tag is bound to the DataGrid control's `dataProvider` property to populate the DataGrid control. The `cellRenderer` property for the second DataGridColumn object contains a binding expression that sets its value to a custom cell renderer named `DirectionRenderer`, which is implemented as an ActionScript class in a file named `DirectionRenderer.as`.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

<!-- Data model -->
  <mx:Model id="stateModel" source="DataProvider.xml"/>

  <!-- User interface -->
  <mx:DataGrid width="100%" height="100%" id="dg"
    dataProvider="{stateModel.dataProvider}">
    <mx:columns>
      <mx:Array>
        <mx:DataGridColumn columnName="name" />
        <mx:DataGridColumn columnName="direction"
          cellRenderer="DirectionRenderer"/>
      </mx:Array>
    </mx:columns>
  </mx:DataGrid>
</mx:Application>
```

The following example shows the MXML code for an application that uses the image cell renderer in a List control instead of a DataGrid control:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

<!-- Data model -->
  <mx:Model id="stateModel" source="DataProvider.xml"/>

  <!-- User interface. -->
  <mx:List width="100%" height="100%" id="list1"
    cellRenderer="DirectionRenderer"
    dataProvider="{stateModel.dataProvider}"/>
</mx:Application>
```

The following file, named `DataProvider.xml`, contains the data source for the `<mx:Model>` tag:

```
<?xml version="1.0" ?>
<stateList>
  <dataProvider name="California" direction="LEFT" />
  <dataProvider name="New York" direction="RIGHT" />
  <dataProvider name="Alberta" />
  <dataProvider name="Texas" direction="DOWN" />
  <dataProvider name="Canada" direction="UP" />
  <dataProvider name="Oakland" />
  <dataProvider name="Kansas" direction="RIGHT" />
  <dataProvider name="Detroit" />
</stateList>
```

Cell renderer code

The following example shows the source code for the `DirectionRenderer` cell renderer class. The class is contained in a file named `DirectionRenderer.as`. The `[Embed...]` metadata tags embed images into the variables that are declared directly below them. The `setValue()` method contains code that determines which of the embedded images to display in each row in the `DataGridColumn` object. The `switch` statement in the `setValue()` method determines the image to display in each row based on the value of the `direction` property of each `<dataProvider... />` tag in the `DataProvider.xml` file.

```
import mx.core.UIComponent
import mx.controls.Image

class DirectionRenderer extends UIComponent
{
  [Embed(source="UP.jpg")]
  var image_up :String;
  [Embed(source="DOWN.jpg")]
  var image_down :String;
  [Embed(source="LEFT.jpg")]
  var image_left :String;
  [Embed(source="RIGHT.jpg")]
  var image_right :String;
  [Embed(source="UNKNOWN.jpg")]
  var image_unknown :String;

  var image : MovieClip;

  function ImageCellRenderer() {
  }

  function createChildren() : Void {
    image = createClassObject( Image, "image", 1, {owner:this});
  }

  function layoutChildren() : Void {
    // Resize images if necessary.
  }

  function setValue( str:String, item:Object, sel:Boolean ) : Void {
```

```

if ( item !== undefined ) {
    var currentImage:String = image.source;
    var newImage:String = "";

    if (image==undefined)
        createClassObject(Image, "image", 1, {owner:this});

    switch ( item.direction ) {
        case "UP":
            newImage = image_up;
            break;
        case "DOWN":
            newImage = image_down;
            break;
        case "LEFT":
            newImage = image_left;
            break;
        case "RIGHT":
            newImage = image_right;
            break;
        default :
            newImage = image_unknown;
            break;
    }

    if ( newImage !== currentImage )
        image.source = newImage;
    } else {
        destroyObject("image");
        delete image;
    }
    size();
}

function getPreferredHeight() : Number {
    return (image!==undefined)?image.height:0;
}

function getPreferredWidth() : Number {
    return (image!==undefined)?image.width:0;
}
}

```

Validating data in a List-based control

Just as you can validate data in other types of controls, you can validate data in the cells of List-based controls. To do so, you can create a cell renderer that incorporates a data validator. For more information about data validators, see [Chapter 31, “Validating Data in Flex,”](#) on page 703.

The following figure shows an application that contains a DataGrid control that uses a validating cell renderer to validate phone number values. The second column uses a cell renderer implemented as an MXML component.

| Name | Phone | Email |
|---------------------|--------------|---------------------------|
| Christina Coenraets | 555-219-2270 | ccoenraets@fictitious.com |
| Louis Freligh | 555-219-2100 | lfreligh@fictitious.com |
| Ronnie Hodgman | 555-219-2030 | rhodgman@fictitious.com |
| Joanne Wall | 555-219-2012 | jwall@fictitious.com |
| Maurice Smith | 2192012 | maurice@fictitious.com |
| Mary Jones | 555-219-2000 | mjones@fictitious.com |
| | | |
| | | |

MXML code

The following example shows the MXML code for the application. The `dataProvider` property of the DataGrid control contains a binding expression that sets its value to data in the `<mx:Model>` tag. The `cellRenderer` property of the first DataGridColumn object contains a binding expression that sets its value to a custom cell renderer named `validatingRenderer`, which is implemented as an MXML component in a file named `validatingRenderer.mxml`.

```
<?xml version="1.0" ?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">

  <mx:Model id="employeeModel" source="employees.xml"/>

  <mx:DataGrid id="dg" width="500" height="200"
    dataProvider="{employeeModel.employee}">
    <mx:columns>
      <mx:Array>
        <mx:DataGridColumn columnName="name" headerText="Name" />
        <mx:DataGridColumn columnName="phone" headerText="Phone"
          cellRenderer="validatingRenderer"/>
        <mx:DataGridColumn columnName="email" headerText="Email" />
      </mx:Array>
    </mx:columns>
  </mx:DataGrid>
</mx:Application>
```

The following example shows the MXML code for an application that uses the validating renderer in a List control instead of a DataGrid control:

```
<?xml version="1.0" ?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:Model id="employeeModel" source="employees.xml"/>
  <mx:List id="list1" width="500" height="200" labelField="phone"
    cellRenderer="{validatingRenderer}"
    dataProvider="{employeeModel.employee}"/>
</mx:Application>
```

The following example is part of a file named `employees.xml` that contains the data source for the

```
<mx:Model> tag:

<?xml version="1.0" ?>
<employees>
<employee>
    <name>Christina Coenraets</name>
    <phone>555-219-2270</phone>
    <email>ccoenraets@fictitious.com</email>
    <active>true</active>
</employee>
<employee>
    <name>Louis Freligh</name>
    <phone>555-219-2100</phone>
    <email>lfreligh@fictitious.com</email>
    <active>true</active>
</employee>
...
</employees>
```

Cell renderer code

The following example shows the source code for the validating renderer that is implemented as an MXML component. The MXML component is a `TextInput` control that implements a `setValue()` method; this is the only method that you must explicitly implement.

In this example, a `ZipCodeValidator` validator is assigned to the `text` property of the `TextInput` control.

```
<?xml version="1.0"?>
<mx:TextInput xmlns:mx="http://www.macromedia.com/2003/mxml">
    <mx:Script>
        <![CDATA[
            function setValue( str, item:Object, sel:Boolean ) {
                if (item==undefined) {
                    this.clear();
                } else {
                    this.text = str;
                }
            }
        ]]>
    </mx:Script>

    <mx:String id="value">{this.text}</mx:String>
    <mx:PhoneNumberValidator field="value" listener="this" />
</mx:TextInput>
```

Formatting data in a List-based control

Just as you can format data in other types of controls, you can format data in the cells of List-based controls. You can apply a formatter to a List-based control without creating a custom cell renderer.

The following figure shows a `DataGridColumn` object that uses a `NumberFormatter` formatter to round the numbers in the second column:

| age | age-Formatter |
|----------|---------------|
| 28.4325 | 28.4 |
| 26.525 | 26.5 |
| 35.4324 | 35.4 |
| 28.23432 | 28.2 |
| | |
| | |

To apply a data formatter to the cells of a List-based control, you do the following:

1. Declare a formatter tag in MXML; for example:

```
<mx:NumberFormatter id="FORMATTER" precision="1"/>
```

2. Specify an `ActionScript` function as the value of a `labelFunction` property of a `List` control or `DataGridColumn` object; for example:

```
<mx>DataGridColumn columnName="age" headerText="age-Formatter"
    labelFunction="formatAge"/>
```

3. Call the formatter's `format()` method in the `ActionScript` function specified in the `labelFunction` property; for example:

```
function formatAge(item) {
    return FORMATTER.format(item.age);
}
```

MXML code

The following example shows the MXML code for the application. The `dataProvider` property of the `DataGrid` control contains a binding expression that sets its value to data in the

`<mx:Model>` tag.

```
<?xml version="1.0" ?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    xmlns:my="renderers" >

    <mx:Model id="dataModel" source="DataProvider.xml"/>

    <mx>DataGrid editable="true" id="dg"
        dataProvider="{dataModel.dataProvider}">
        <mx:columns>
            <mx:Array>
                <mx>DataGridColumn columnName="age"/>
                <mx>DataGridColumn columnName="age" headerText="age-Formatter"
                    labelFunction="formatAge"/>
            </mx:Array>
        </mx:columns>
    </mx>DataGrid>

    <mx:NumberFormatter id="FORMATTER" precision="1"/>
```



```
<mx:Script>
  <![CDATA[
    function formatAge(item){
      return FORMATTER.format(item.age);
    }
  ]]>
</mx:Script>
</mx:Application>
```

The following file, named `DataProvider.xml`, contains the data source for the `<mx:Model>` tag:

```
<?xml version="1.0" ?>
<dataProvider>
  <dataProvider age="28.4325"/>
  <dataProvider age="26.525"/>
  <dataProvider age="35.4324"/>
</dataProvider>
```


CHAPTER 16

Using Styles and Fonts

Styles are useful for defining the look and feel (appearance) of your Macromedia Flex applications. You can use them to change the appearance of a single component, or apply them across all components. This chapter describes how to use styles, including the Cascading Style Sheet (CSS) syntax, in your applications.

Contents

| | |
|---|-----|
| About styles | 395 |
| Using local style definitions | 413 |
| Using the StyleManager | 417 |
| Using the <code>setStyle()</code> and <code>getStyle()</code> methods | 419 |
| Using inline styles | 421 |
| About fonts | 422 |

About styles

You modify the appearance of Flex components through style properties. These properties can define the size of a font used in a Label control, or the background color used in the Tree control. In Flex, some styles are inherited from parent containers to their children, and across style types and classes. This means that you can define a style once, and then have that style apply to all controls of a single type or to a set of controls. In addition, you can override individual properties for each control at a local, document, or global level, giving you great flexibility in controlling the appearance of your applications.

This section introduces you to applying styles to controls. It also provides a primer for using Cascading Style Sheets (CSS), an overview of the style value formats (Length, Color, and Time), and describes style inheritance. Subsequent sections provide detailed information about different ways of applying styles in Flex.

Using styles in Flex

There are many ways to apply styles in Flex. Some provide more granular control and can be approached programmatically. Others are not as flexible, but can require less computation. In Flex, you can apply styles to controls in several ways.

When applying styles, you must be aware of which properties your theme supports. The default theme in Flex does not support all style properties. For more information, see [“About supported styles” on page 410](#).

External style sheets

Use Cascading Styles Sheets (CSS) to apply styles to a document or across entire applications. You can point to a style sheet without invoking `ActionScript`. This is the most concise method of applying styles, but can also be the least flexible. Style sheets can define global styles that are inherited by all controls, or individual classes of styles that only certain controls use.

The following example applies the external style sheet `myStyle.css` to the current document:

```
<mx:Style source="myStyle.css"/>
```

Flex includes a global style sheet defined in the `flex-config.xml` that is the basis of all style definitions applied to all applications. By default, this global style sheet has no style definitions, but it provides a convenient location to define them.

Local style definitions

Use the `<mx:Style>` tag to define styles that apply to the current document and its children. You define styles in the `<mx:Style>` tag using CSS syntax and can define styles that apply to all instances of a control or to individual controls. The following example defines a new style and applies it to only the `myButton` control:

```
<mx:Style>
    .myFontStyle { fontSize: 15 }
</mx:Style>
<mx:Button id="myButton" styleName="myFontStyle" label="Click Here" >
```

The following example defines a new style that applies to all instances of the `Button` class:

```
<mx:Style>
    Button { fontSize: 15 }
</mx:Style>
<mx:Button id="myButton" label="Click Here" >
```

For more information on using local style definitions, see [“Using local style definitions” on page 413](#).

StyleManager class

Use the `mx.styles.StyleManager` class to apply styles to all classes or all instances of specified classes. The following examples set the `fontSize` style to 15 on all `TextArea` controls:

```
StyleManager.styles.TextArea.fontSize = 15;
StyleManager.styles.TextArea.setStyle("fontSize",15);
```

You can also use the `StyleManager` to apply global styles. For more information on using the `StyleManager`, see [“Using the StyleManager” on page 417](#).

getStyle() and setStyle() methods

Use the `setStyle()` and `getStyle()` methods to manipulate style properties on instances of controls. Using these methods to apply styles requires a greater amount of processing power on the client than using style sheets but provides more granular control over how styles are applied.

The following example sets the `fontSize` to 15 on only the `myButton` instance:

```
myButton.setStyle("fontSize", 15);
```

For more information on using the `getStyle()` and `setStyle()` methods, see [“Using the setStyle\(\) and getStyle\(\) methods” on page 419](#).

Inline styles

Use attributes of MXML tags to apply style properties. These properties apply only to the instance of the control. This is the most efficient method of applying instance properties because no `ActionScript` code blocks or method calls are required.

The following example sets the `fontSize` to 15 on the `myButton` instance:

```
<mx:Button id="myButton" fontSize="15" label="My Button"/>
```

For more information on using inline styles, see [“Using inline styles” on page 421](#).

Setting global styles

Most text and color styles, such as `fontSize` and `color`, are inheritable. When you apply an inheritable style to a container, all the children of that container inherit the value of that style property. If you set the `color` of a `Panel` container to green, all buttons in the `Panel` container will also be green, unless those buttons override that color.

Many styles, however, are not inheritable. If you apply them to a parent container, only the container uses that style. Children of that container do not use the values of noninheritable styles.

By using global styles, you can apply noninheritable styles to all controls that do not explicitly override that style. Flex provides the following ways to apply styles globally:

- `StyleManager` `global` style
- CSS `global` type selector

The `StyleManager` lets you apply styles to all controls using the `global` style. For more information on using the `StyleManager` class, see [“Using the StyleManager” on page 417](#).

You can also apply global styles using the `global` type selector in your CSS style definitions. These are located either in external CSS style sheets or in an `<mx:Style>` tag. For more information, see [“Using the global type selector” on page 415](#).

About style value formats

Style properties can be of types Boolean, String, or Number. They can also be arrays of these types. In addition to a type, style properties also have a format (Length, Time, or Color) that describes the valid values of the property. This section describes these formats.

Length format

The Length format applies to any style property that takes a size value, such as the size of a font. Length is of type Number.

The Length type takes the following form:

`[+|-]length[unit]`

Note: Spaces are not allowed between the modifier (+ or -), the length value, and the unit.

The following table describes the Length units:

| Unit | Scale | Description |
|------|----------|--|
| em | Relative | Ems. The width of the character <i>m</i> in the character set. |
| ex | Relative | x-height. The height of the character <i>x</i> in the character set. |
| px | Relative | Pixels. |
| in | Absolute | Inches. |
| cm | Absolute | Centimeters. |
| mm | Absolute | Millimeters. |
| pt | Absolute | Points. |
| pc | Absolute | Picas. |

In Flex, all lengths are converted to pixels prior to being displayed. In this conversion, Flex assumes that an inch equals 72 pixels. All other lengths are based on that assumption. For example, 1 cm is equal to 1/2.54 of an inch. To get the number of pixels in 1 cm, multiply 1 by 72, and divide by 2.54.

There are two categories of units: relative and absolute. Relative values are calculated based on other length units, and, therefore, are more useful when the size and format of the output device is unknown. When you specify relative values, you can use the modifiers + and -.

Absolute values give you more control over specifying the lengths, but do not necessarily scale well to the output device. As a result, relative units are more commonly used in style definitions and produce more predictable results across the most platforms.

When you use inline styles, Flex ignores units and uses pixels as the default.

The `fontSize` style property allows a set of keywords in addition to numbered units. You can use the following keywords when setting the `fontSize` style property. The exact sizes are defined by the client browser.

- `xx-small`
- `x-small`
- `small`
- `medium`
- `large`
- `x-large`
- `xx-large`

The following example class selector defines the `fontSize` as `x-small`:

```
.smallFont {
    fontFamily: Arial, Helvetica, "_sans";
    fontSize: x-small;
    fontStyle: oblique;
}
```

Time format

You use the `Time` format for component properties that move or have built-in effects, such as the `ComboBox` component when it drops down and pops up. The `Time` format is of type `String` and is represented in milliseconds. Do not specify the units when entering a value in the `Time` format.

The following example sets the `selectionDuration` style property of the `myTree` control to 100 milliseconds:

```
myTree.setStyle("selectionDuration", 100);
```

Color format

You define `Color` in several formats. You can use most of the formats only in the CSS style definitions. The following table describes the recognized `Color` formats for a style property:

| Format | Description |
|-------------|---|
| hexadecimal | <p>Hexadecimal colors are represented by a six-digit code preceded by either a zero and small x (0x) or a pound sign (#). The range of possible values is 0x000000 to 0xFFFFFF (or #000000 to #FFFFFF).</p> <p>You can use the 0x prefix when defining colors anywhere. You can use the # prefix in CSS style sheets and in <code><mx:Style></code> tag blocks. Styles that use the # prefix require less client side processing.</p> <p>When using the 0x prefix, you should surround the Hex color value with quotation marks.</p> <p>You can use hexadecimal format in all types of style definitions: inline properties, CSS files, <code><mx:Style></code> tag definitions, <code>setStyle()</code> method calls, and <code>StyleManager</code> definitions.</p> |

| Format | Description |
|-----------------|--|
| RGB | RGB colors are a mixture of the colors red, green, and blue, and are represented in percentages. The format of RGB colors is x%, y%, z%. You can use the RGB format only in style sheet definitions. |
| 8-bit octet RGB | The 8-bit octet RGB colors are red, green, and blue values from 1 to 255. The format of 8-bit octet colors is [0-255], [0-255], [0-255]. You can use the 8-bit octet RGB format only in style sheet definitions. |
| VGA color names | VGA color names are a set of 16 basic colors supported by all browsers that support CSS. The available color names are Aqua, Black, Blue, Fuchsia, Gray, Green, Lime, Maroon, Navy, Olive, Purple, Red, Silver, Teal, White, Yellow. Some browsers support a larger list of color names. You can use the VGA color names format in style sheet definitions and inline style declarations. VGA color names are not case-sensitive. Do not surround color names with quotation marks in CSS definitions. |

Color formats are of type Number. When you specify a format such as a VGA color name, Flex converts that String to a Number.

CSS style definitions and the `<mx:Style>` tag support the four color formats, as the following example shows:

```
<mx:Style>
    .myclass {
        shadowColor: #6666CC;           // CSS hexadecimal format
        fillColor: "0x6666CC";         // Hexadecimal format
        borderColor: rgb(77%,22%,0%);   // RGB format
        errorColor: rgb(0,255,0);        // 8-bit octet RGB format
        color: Blue;                    // VGA color name
    }
</mx:Style>
```

The `StyleManager` and `setStyle()` method support only the hexadecimal color format. You can optionally surround the value with quotation marks, as the following example shows:

```
StyleManager.styles.TextArea.setStyle("color", "0xFF0099");
btn2.setStyle("color", "0x999933");
```

When setting style properties inline, you can use either the hexadecimal format or the VGA color name, as the following example shows:

```
<mx:Button id="btn1" label="Click 1" color="0x9966CC"/>
<mx:Button id="btn2" label="Click 2" color="Yellow"/>
```

When defining styles in CSS style sheets or in an `<mx:Style>` tag, you should use the `#` prefix rather than the `0x` prefix in the color definition, as the following example shows:

```
.myStyle, Button {
    color: #FF0033; // preferred
    color: 0xFF0033; // not preferred
}
```

This method of assigning a color value is more efficient than using the `0x` prefix because it uses a CSS color value. A string value must be interpreted into a color value before it can be applied.

Some controls accept arrays of colors. For example, the `Tree` control's `depthColors` style property can use a different background color for each level in the tree. To assign colors to a property in an array, add the items in a comma-separated list to the property's definition. The index is assigned to each entry in the order that it appears in the list.

The following example defines arrays of colors for properties of the `Tree` type selector:

```
Tree {
    depthColors: #EAEAEA, #FF22CC, #FFFFFF;
    alternatingRowColors: red, green, blue, yellow;
}
```

In addition to defining properties that take an array of values using a style sheet, you can define the array of an instance property in ActionScript using a comma-separated list of values, as the following example shows:

```
myTree.setStyle("depthColors",[0xEAEAEA, 0xFF22CC, 0xFFFFFFFF]);
```

You can also set the `depthColors` property inline, as the following sample application shows:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">

    <mx:XML id="dp">
        <node label="New">
            <node label="HTML Document" />
            <node label="Text Document" />
        </node>
        <node label="Close"/>
    </mx:XML>

    <mx:Tree depthColors="[0xEFEFEF, 0xFF0000]" dataProvider="{dp}"
        rollOverColor="0xFFFFFFFF"/>

</mx:Application>
```

The syntax for setting an array of colors with the `setStyle()` method is similar to setting inline properties, except that you do not surround the square brackets with quotation marks. The following example sets the `fillColors` property:

```
myAccordion.setStyle("fillColors",[0xFFCC66,0x00CC00]);
```

About Cascading Style Sheets

Cascading Style Sheets (CSS) are a standard mechanism for declaring text styles in HTML and most scripting languages. A style sheet is a collection of formatting rules for types of components or classes that include sets of components. Flex supports the use of CSS syntax and styles to apply styles to Flex components.

In CSS syntax, each declaration associates a style name, or *selector*, with one or more style properties and their values. For example, the following style defines a selector named `bodyText`.

```
.bodyText { textAlign: left }
```

In this example, `bodyText` defines a new class of styles, so it is called a *class selector*. In the markup, you can explicitly apply the `bodyText` style to a control.

A *type selector* implicitly applies itself to all components of a particular type. The following example defines a type selector named `Label`:

```
Label { textAlign: left }
```

Flex applies this style to all components of type `Label`.

Note: The names of class selectors cannot include hyphens in Flex. If you use a hyphenated class selector name, such as `my-class-selector`, Flex ignores the style.

You define multiple style properties in each selector by separating each property with a semicolon, as the following example shows:

```
Label {
    textAlign: left;
    fontSize: 12;
    color: Blue;
}
```

You can programmatically define values in class and type selectors using the `mx.styles.StyleManager` class. For more information, see [“Using the StyleManager” on page 417](#).

Applying color formats in CSS

CSS style definitions support all four color formats, as the following example shows:

```
<mx:Style>
    .myClass { fillColor: #6666CC }           // Hexadecimal format
    .yourClass { borderColor: rgb(77%,22%,0%) } // RGB format
    .hisClass { errorColor: rgb(0,255,0) }     // 8-bit octet RGB format
    .herClass { color: Blue }                 // VGA color name
</mx:Style>
```

When using hexadecimal color values in CSS style sheets or in an `<mx:Style>` tag, you should use the `#` prefix rather than the `0x` prefix in the color definition, as the following example shows:

```
.myStyle, Button {
    color: #FF0033; // Preferred
    color: 0xFF0033; // Not preferred
}
```

This method of assigning a color value is more efficient than using the `0x` method because it uses a CSS color value. A string value must be interpreted into a color value before it can be applied.

About inheritance in CSS

Some style properties are inherited. If you set an inheritable style property on a parent container, its children inherit that style property. For example, if you define `fontFamily` as `Times` for a `Panel` container, all children of that container will also use `Times` for `fontFamily`, unless they override that property.

In general, color and text styles are inheritable, regardless of how they are set (using CSS or style properties). All other styles are not inheritable unless otherwise noted.

If you set a noninheritable style such as `textDecoration` on a parent container, only the parent container and not its children use that style. For more information on inheritable style properties, see [“About style inheritance” on page 407](#).

There is an exception to the rules of inheritance. If you use the `global` type selector in a CSS style definition, Flex applies those style properties to all controls, regardless of whether the properties are inheritable. For more information about the `global` type selector, see [“Using the global type selector” on page 415](#).

CSS differences

There are two major differences in Flex between support of CSS and the CSS specification:

- Flex supports a subset of the style properties that are available in CSS. Flex controls also have unique style properties that are not defined by the CSS specification. For a list of styles that you can apply to your Flex controls, see [“Supported CSS properties” on page 406](#).
- Flex controls support styles that are defined by the current theme. If a theme does not use a particular style, applying that style to a control or group of controls has no effect. For example, the default theme, Halo, does not support styles such as `symbolColor` and `symbolBackgroundColor`. For more information on themes, see [“About themes” on page 432](#).

About class selectors

Class selectors define a set of styles (or a class) that you can apply to any component. You define the style class, and then point to the style class using the `styleName` property of the component’s MXML tag. All Flex components that are a subclass of the `UIComponent` class support the `styleName` property.

The following example defines a new style `myclass` and applies that style to a `Button` component by assigning the `Button` to the `myclass` style class:

```
<mx:Style>
    .myclass { color: #6666CC }
</mx:Style>
<mx:Canvas>
    <mx:Button styleName="myclass" label="This text is dark blue">
</mx:Canvas>
```

About type selectors

Type selectors assign styles to all components of a particular type. When you define a type selector, you are not required to explicitly apply that style. Instead, Flex applies the style to all classes of that type.

The following example shows a type selector for the `Button` component:

```
<mx:Style>
    Button { color: #6666CC } // Dark blue
</mx:Style>
<mx:Canvas>
    <mx:Button label="This text is dark blue">
</mx:Canvas>
```

In this example, Flex applies the `color` style to all Button components in the current document, and all Button controls in all the child documents.

You can set the same style declaration for multiple component types by using a comma-separated list of components. The following example defines style information for all Buttons, Labels, and TextInput components:

```
<mx:Style>
    Button, TextInput, Label { fontStyle: italic }
</mx:Style>
```

Flex does not support contextual or sequential selectors.

You can use multiple type selectors of the same name at different levels to set different style properties. In a global CSS file, you can set all Label components to use the Blue color for the fonts, as the following example shows:

```
Label { color: Blue }
```

Then, in a local style declaration, you can set all Labels to use the font size 10, as the following example shows:

```
<mx:Style>
    Label { fontSize: 10pt }
</mx:Style>
```

The local style declaration does not interfere with the global style declaration. Flex applies only the style properties that you specified. The result of this example is that Label controls that are children of the current document use Blue for the color and 10 for the font size.

Global styles are shared across all documents in an application and across all applications that are loaded inside the same application. For example, if you load two SWF files inside separate tabs in a TabNavigator container that uses Loader controls, both SWF files share the global style definitions in the parent application and each other.

Using compound selectors

You can mix class and type selectors to create a component that has styles based on compound style declarations. For example, you can define the color in a class selector and the font size in a type selector, and then apply both to the component:

```
<mx:Style>
    Label { fontSize: 10pt }
    .myLabel { color: Blue }
</mx:Style>
<mx:Label styleName="myLabel" label="This label is 10pt Blue">
    ...
</mx:Label>
```

About selector precedence

Class selectors take precedence over type selectors. In the following example, the text for the first button (with the class selector) is red, and the text of the second button (with the implicit type selector) is yellow:

```
<mx:VBox xmlns:mx="http://www.macromedia.com/2003/mxml" width="500"
  height="200">
  <mx:Style>
    .myclass { color: Red }
    Button { fontSize: 10pt; color: Yellow }
  </mx:Style>
  <mx:Button styleName="myclass" label="I am red"></mx:Button>
  <mx:Button label="I am yellow"></mx:Button>
</mx:VBox>
```

The font size of both buttons is 10. When a class selector overrides a type selector, it does not override all values, just those that are explicitly defined.

Default application style

Flex applications have default style settings that define the application's appearance in a browser. The following table lists the default style settings:

| Style | Default |
|-----------------|---------------|
| backgroundImage | Gray gradient |
| backgroundSize | 100% |
| marginTop | 24 pixels |
| marginLeft | 24 pixels |
| marginBottom | 24 pixels |
| marginRight | 24 pixels |
| horizontalAlign | Centered |

You can override these settings with a built-in style: `plain`. To use `plain`, you reference it in the `<mx:Application>` tag's `styleName` property, as the following example shows:

```
<mx:Application styleName="plain" />
```

The following table lists the properties of the `plain` style:

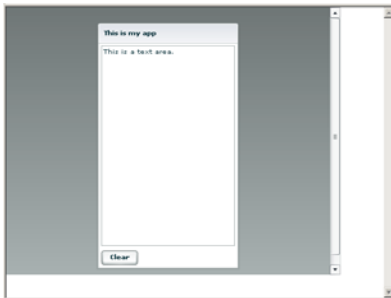
| Style | Description |
|-----------------|-------------------|
| backgroundImage | White (0xFFFFFF). |
| backgroundSize | None. |
| marginTop | 0 pixels. |
| marginLeft | 0 pixels. |
| marginBottom | 0 pixels. |

| Style | Description |
|-----------------|-------------|
| marginRight | 0 pixels. |
| horizontalAlign | Left. |

The `plain` style does not change the default stage color, which is the color of the background when the initialize progress bar appears. To change this, set the `<mx:Application>` tag's `backgroundColor` property to `0xFFFFFF`, as the following example shows:

```
<mx:Application styleName="plain" backgroundColor="0xFFFFFF" />
```

The following image shows the differences between the default style and an application that applies the `plain` style:



Default application style



Plain application style

Supported CSS properties

Flex supports the following subset of the CSS style properties as defined by the CSS specification:

- `color`
- `display`
- `fontFamily`
- `fontSize`
- `fontStyle`
- `fontWeight`
- `marginLeft`
- `marginRight`
- `textAlign`
- `textDecoration`
- `textIndent`

Flex also supports properties that you can define using CSS syntax and inheritance rules, but are not part of the CSS property library.

You cannot bind style properties.

Using embedded resources in style sheets

You can use embedded resources in your style sheets. This is useful for style properties such as `backgroundImage`, which you can apply to an embedded resource such as an image file.

The following example embeds an image as `myImage`. The style declaration references this resource:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:Style>
    .style1 { backgroundImage : "myImage" }
  </mx:Style>
  <mx:Script>
    [Embed(source="images/image1.gif")]
    var myImage:String;
  </mx:Script>
  <mx:HBox>
    <mx:TextArea width="200" height="200" styleName="style1"/>
  </mx:HBox>
</mx:Application>
```

About style inheritance

If you define a style in only one place in a document, Flex uses that definition to set a property's value. However, an application can have several style sheets, local style definitions, global style properties, and style properties set directly on component instances. In such a situation, Flex determines the value of a property by looking for its definition in all these places in a specific order.

Lower-level styles take precedence over higher level or global styles. If you set a style on an instance and then set the style globally, the global style does not override the local style, even if you set it after you set the local style.

Style inheritance order

The order in which Flex looks for styles is important to understand so that you can know which style properties apply to which controls.

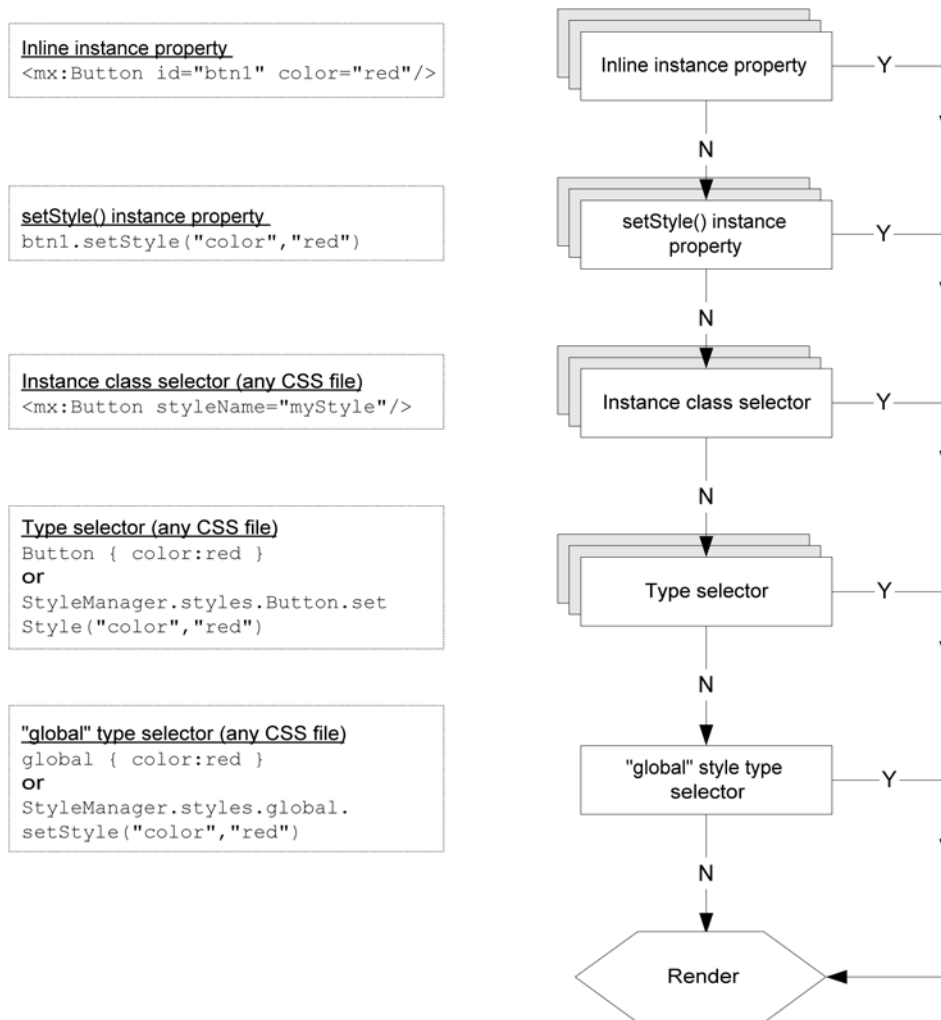
Flex looks for a style property that was set inline on the component instance. If no style was set on the instance using an inline style, Flex checks if a style was set using an instance's `setStyle()` method. If did not directly set the style on the instance, Flex examines the `styleName` property of the instance to see if a style declaration is assigned to it.

If you did not assign the `styleName` property to a style declaration, Flex looks for the property on a type selector style declaration. If there are no type selector declarations, Flex checks the global style declaration. If all of these checks fail, the property is undefined, and Flex applies the default style.

In the early stages of checking for a style, Flex also examines the control's parent container for style settings. If the style property is not defined and the property is inheritable, Flex looks for the property on the instance's parent container. If the property isn't defined on the parent container, Flex checks the parent's parent, and so on. If the property is not inheritable, Flex ignores parent container style settings.

Flex does not examine the parent *class* of a container to determine style information, but rather, it examines *instances* of the parent class. For example, if an MXML component's root element is Panel, and no style for the type Panel is in that document, Flex does not use styles for the Container type (mx.containers.Panel extends mx.containers.Container).

The following figure shows the flow of the Flex style assignment operation:



In this image, the shaded boxes indicate where Flex checks if the parent container's style property was set. If Flex finds a setting on a parent container, and the style is inheritable, Flex immediately stops checking for styles and renders the control.

Style definitions in `<mx:Style>` tags, external style sheets, and the `global.css` style sheet also follow an order of precedence. The same style definition in `global.css` is overridden by an external style sheet specified by an `<mx:Style source="stylesheet"/>` tag, which is overridden by a style definition within an `<mx:Style>` tag.

The following example defines a type selector for `Panel` that sets the `fontFamily` property to `Times`. As a result, all controls inside the `Panel` container inherit that style. However, `button2` overrides the inherited style by defining the `fontFamily` style inline. When the application renders, `button2` uses `Arial` for the font.

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" width="500"
  height="400" >
  <mx:Style>
    Panel {
      fontFamily: Times, "_serif";
    }
  </mx:Style>
  <mx:Panel title="My Panel">
    <mx:Button id="button1" label="Button 1" />
    <mx:Button id="button2" label="Button 2" fontFamily="Arial" />
    <mx:TextArea text="Flex has its own set of style properties which are
    extensible so you can add to that list when you create a custom component."
    width="425" height="400" />
  </mx:Panel>
</mx:Application>
```

Inheritance exceptions

Not all styles are inheritable, and not all styles are supported by all components and themes. In general, color and text styles are inheritable, regardless of how they are set (using CSS or style properties). All other styles are not inheritable unless otherwise noted.

A style is inherited only if it meets the following conditions:

- The style is supported by the theme. For a list of styles supported by the default Flex theme, see [“About supported styles” on page 410](#).
- The style is inheritable. For a list of styles and their inheritance, see *Flex ActionScript and MXML API Reference*.
- The style is supported by the control. For information about which controls support which styles, see the control's description in [Chapter 2, “Using Controls,” on page 31](#).
- The style is set on the control's parent container or the container's parent. A style doesn't get inherited from another class, unless that class is a parent container of the control, or a parent container of the control's parent container.

- The style is not overridden at a lower level. For example, if you define a style type selector (such as `Button { color:red }`), but then set an instance property on a control (such as `<mx:Button color="blue"/>`), the type selector style will not override the style instance property even if the style is inheritable.

You can apply noninheritable styles to all controls using the `global` type selector. For more information, see [“Using the global type selector” on page 415](#).

About supported styles

All themes support the inheritable and noninheritable text styles, but not all styles are supported by all themes. If you try to set a style property on a control but the current theme does not support that style, Flex does not apply the style.

Some styles are only used by skins in the theme, while others are used by the component code itself. The display text of components is not skinnable, so support for text styles is theme-independent.

All themes support the following inheritable text styles:

- `color`
- `fontFamily`
- `fontSize`
- `fontStyle`
- `fontWeight`
- `textAlign`
- `textIndent`

All themes support the following noninheritable text styles:

- `marginLeft`
- `marginRight`
- `textDecoration`

All themes support the following component styles (these are component-defined, not skin-defined):

- `direction`
- `horizontalAlign`
- `horizontalGap`
- `leading`
- `marginLeft`
- `marginRight`
- `marginTop`
- `marginBottom`
- `headerHeight`
- `verticalAlign`

- `verticalGap`

In addition to the text styles, the Halo theme supports the following styles (not all of these styles are available on all components):

- `borderColor`
- `borderSides`
- `borderThickness`
- `cornerRadius`
- `dateHeaderColor`
- `dateRollOverColor`
- `direction`
- `disabledColor`
- `dropShadow`
- `fillColors`
- `footerColors`
- `headerColors`
- `headerHeight`
- `highlightColor`
- `horizontalAlign`
- `horizontalGap`
- `leading`
- `marginLeft`
- `marginRight`
- `marginTop`
- `marginBottom`
- `panelBorderStyle`
- `rollOverColor`
- `selectionColor`
- `selectedDateColor`
- `shadowCapColor`
- `shadowColor`
- `shadowDirection`
- `shadowDistance`
- `strokeWidth`
- `tabHeight`
- `tabWidth`
- `textRollOverColor`
- `textSelectedColor`

- `themeColor`
- `todayColor`
- `verticalAlign`
- `verticalGap`

For more information about the Halo theme and other sample themes included with Flex, see [Chapter 17, “Using Themes and Skins,” on page 429](#).

Using external style sheets

Flex supports external CSS style sheets. You can declare the location of a local style sheet or use the global style sheet to define the styles that all applications use. To apply a style sheet to the current document and its child documents, use the `source` property of the `<mx:Style>` tag.

Note: You should try to limit the number of style sheets used in an application, and set the style sheet only at the top-level document in the application (the document that contains the `<mx:Application>` tag). If you set a style sheet only on a child document, unexpected results can occur.

The following example points to the `MyStyleSheet.css` file in the `flex_app_root/assets` directory:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" width="500"
  height="500">
  <mx:Style source="/assets/MyStyleSheet.css"/>
  ...
</mx:Application>
```

The value of the `source` property is the URL of a file that contains style declarations. When you use the `source` property, the contents of *that* `<mx:Style>` tag must be empty. You can use additional `<mx:Style>` tags to define other styles. Do not add `<mx:Style>` tags to your included file.

The external style sheet file can contain both type and class selectors. External style sheets support the four `Color` formats, as the following example shows:

```
.myclass { fillColor: #6666CC } // Hexadecimal format
.myclass { borderColor: rgb(77%,22%,0%) } // RGB format
.myclass { errorColor: rgb(0,255,0) } // 8-bit octet RGB format
.myclass { color: Blue } // VGA color name
```

Using the global style sheet

Flex includes a default style sheet that is used across all applications. You can use it to apply a consistent theme for your applications. The default `global.css` file is empty.

You specify the location of the global style sheet in the `flex_app_root/WEB-INF/flex/flex-config.xml` file using the `<global-css-url>` child tag of the `<compiler>` tag. The default style sheet is defined as follows:

```
<compiler>
...
  <global-css-url>/WEB-INF/flex/global.css</global-css-url>
...
</compiler>
```

If you specify a relative URL, such as one that begins with a forward slash, the value of the `<global-css-url>` tag is relative to the application root. You can specify an absolute URL that points to a style sheet in another domain, as the following example shows:

```
<global-css-url>http://www.acme.com/styles.css</global-css-url>
```

Adding a style definition to the `global.css` file is not the same as applying a global style. It acts as a basis from which Flex builds the application's runtime style sheet. You can use the `global` type selector in an external style sheet to apply noninheritable styles to all controls. For more information, see [“Using the global type selector” on page 415](#).

Using local style definitions

The `<mx:Style>` tag contains style sheet definitions that adhere to the CSS 2.0 syntax. These style sheets apply to the current document and all children of the current document. The `<mx:Style>` tag has the following syntax to define local styles:

```
<mx:Style>
    selector_name {
        style_property: value;
        [...]
    }
</mx:Style>
```

The following example defines a class and a type selector in the `<mx:Style>` tag:

```
<mx:Style>
    .myclass { color: Red } /* class selector */
    Button { fontSize: 10pt; color: Yellow } /* type selector */
</mx:Style>
```

For ActionScript, the naming convention for property names is to use mixed case. For CSS properties, the convention is to use a hyphen. In style definitions, you can use either the ActionScript property name or the CSS property names, as the following example shows:

```
.myclass { fontStyle: italic } /* Valid property name */
.myclass { font-style: italic } /* Valid property name */
Button { fontSize: 14 } /* Valid property name */
Button { font-size: 14 } /* Valid property name */
```

However, for the style name itself, you cannot use a hyphenated name, as the following example shows:

```
.myClass { ... } /* Valid style name */
.my-class { ... } /* Not a valid style name */
```

Local style definitions support all four color formats, as the following example shows:

```
<mx:Style>
    .myClass { color: #6666CC } // Hexadecimal format
    .yourClass { color: rgb(77%,22%,0%) } // RGB format
    .hisClass { color: rgb(0,255,0) } // 8-bit octet RGB format
    .herClass { color: Blue } // VGA color name
</mx:Style>
```

Using the Application type selector

The Application container is the top-most container in a Flex application. Styles defined on the Application type selector that are inheritable are inherited by all of the container's children. Styles that are not inheritable are only applied to the Application container itself and not its children.

Styles applied with the Application type selector are not inherited by the Application object's children if those styles are noninheritable. To use CSS to apply a noninheritable style globally, you can use the `global` type selector. For more information, see [“Using the global type selector” on page 415](#).

When you define the styles for the Application type selector, you are not required to declare a style for each component, because the components are children of these classes and inherit the Application type selector styles.

Use the following syntax to define styles for the Application type selector:

```
<mx:Style>
    Application { style_definition }
</mx:Style>
```

The following example defines the Application type selector's `fontFace` and `fontSize`. Flex applies this style to all components in the application that have the `fontFace` and `fontSize` styles; in this case, the Button, Label, and TextField controls.

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" width="425"
    height="400">
    <mx:Style>
        @font-face { fontFamily:myfont; src:url("fonts/AGENCYR.TTF"); }
        Application { fontFamily:myfont; fontSize: 18pt }
    </mx:Style>
    <mx:Script><![CDATA[
        function detectFontType() {
            var f2 = String(application.isFontEmbedded("myfont"));
            return f2;
        }
    ]]></mx:Script>
    <mx:Button label="Click Me" id="btn" click="f1.text=detectFontType();"/>
    <mx:Label styleName="mystyle" id="out" text="Result" />
    <mx:TextArea width="400" height="75" id="f1" text="" />
</mx:Application>
```

In addition to using the Application type selector to define styles, you can create a custom style sheet inside an `<mx:Style>` tag, and then attach that style sheet to the Application object using its `styleName` property, as the following example shows:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" width="425"
    height="400" styleName="myGlobalStyle" >
    <mx:Style>
        myGlobalStyle { fontFamily:myfont; fontSize: 18pt }
    </mx:Style>
    ...
</mx:Application>
```

You can use the Application type selector to set the background image and other display settings that define the way the Flex application appears in a browser. The following sample Application style definition aligns the SWF file to the left, removes margins, and sets the background image to be empty:

```
Application {
    marginLeft: 0px;
    marginRight: 0px;
    marginTop: 0px;
    marginBottom: 0px;
    horizontalAlign: "left";
    backgroundImage: " "; // Empty string sets the image to nothing.
}
```

You can programmatically define values in the Application type selector using the `mx.styles.StyleManager` class. For more information, see [“Using the StyleManager” on page 417](#).

Using the global type selector

Flex includes a `global` type selector that you can use to apply noninheritable styles to all controls. Properties defined by a `global` type selector apply to every control unless that control explicitly overrides it.

The following example defines `fontSize`, an inheritable property, and `textDecoration`, a noninheriting property, to the `global` type selector:

```
global {
    fontSize:22;
    textDecoration: underline;
}
```

Defining styles for complex components

Some complex components include multiple subcontrols. These components often have properties that apply a style class definition to each subcontrol. To style complex components, Flex lets you apply style definitions to the subcontrols rather than the main control using a *style declaration* property. You can also apply global styles with the `StyleManager` or as a `global` type selector, and those styles apply to the subcontrols of complex components.

For example, the `Alert` control comprises a title bar, message area, and set of buttons. You can add styles to the title, message, and buttons by using the `titleStyleDeclaration`, `messageStyleDeclaration`, and `buttonStyleDeclaration` properties of the `Alert` control. Other complex controls that have style declaration properties include `TitleWindow`, `DateChooser`, `DateField`, `TabBar`, and `TabNavigator`. The `Panel` container also has style declaration properties. Most controls and containers do not have style declaration properties. For information, see the control's or container's entry in [Chapter 2, “Using Controls,” on page 31](#), [Chapter 6, “Using Layout Containers,” on page 205](#), or [Chapter 7, “Using Navigator Containers,” on page 247](#).

For `Panel`, `TitleWindow` and other containers and controls, you declare style declaration properties as instance properties in the MXML tag. For `Alert`, you set style declaration properties as static class properties in `ActionScript`, because you cannot set individual properties on an `Alert`.

The following example defines three class styles in an `<mx:Style>` block and applies these to the Alert control's style declaration properties in the `initAlert()` method. The result is that the Alert control's title bar, message area, and buttons each have a unique style:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
  initialize="initAlert()" width="700" height="1000">
  <mx:Script><![CDATA[
    import mx.controls.Alert;
    function initAlert() {
      Alert.titleStyleDeclaration = "AlertTitle";
      Alert.messageStyleDeclaration = "AlertMessage";
      Alert.buttonStyleDeclaration = "AlertButton";
    }
  ]]></mx:Script>
  <mx:Style>
    AlertTitle { color: red; font-family: Symbol; font-size: 16pt;
      font-style: italic; font-weight: bold; }
    AlertMessage { color: blue; font-family: Verdana; font-size: 8pt;
      font-style: italic; font-weight: bold; backgroundColor: yellow; }
    AlertButton { color: green; font-family: Georgia; font-size: 12pt;
      font-style: italic; font-weight: bold; }
  </mx:Style>
  <mx:Grid>
    <mx:GridRow>
      <mx:GridItem>
        <mx:Button id="showAlert_btn" label="show Alert dialog"
          click="alert('This tests style objects!', 'Test Styles',
            Alert.OK | Alert.CANCEL | Alert.NONMODAL)" />
      </mx:GridItem>
    </mx:GridRow>
  </mx:Grid>
</mx:Application>
```

The following example defines a the `PanelTitle` style in an `<mx:Style>` block. The Panel container's MXML tag assigns the `titleStyleDeclaration` property to the `PanelTitle` style. The result is that the Panel's title text is green:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" width="500"
  height="400">
  <mx:Style>
    PanelTitle {
      color: green;
    }
  </mx:Style>
  <mx:Panel title="My Panel" titleStyleDeclaration="PanelTitle" width="200">
    // Panel contents
  </mx:Panel>
</mx:Application>
```


Using the StyleManager

The `mx.styles.StyleManager` class lets you access global style sheets, class selectors, and type selectors in ActionScript. It also lets you apply inheritable and noninheritable properties globally. Using the StyleManager, you can define new CSS style declarations and apply them to controls in your Flex applications.

To set a value using the StyleManager, use the following syntax:

```
mx.styles.StyleManager.styles.style_name.setStyle("property", value);
```

The `style_name` can be the global style sheet (the literal “global”), a type selector such as `Button` or `TextArea`, or a class selector that you define in either the `<mx:Style>` tag or an external style sheet. In addition, it can be any object of type `CSSStyleDeclaration`.

You can also set the value of a style property directly using the following syntax:

```
mx.styles.StyleManager.styles.style_name.property = "value";
```

The former method is preferable because calling the `setStyle()` method forces the Macromedia Flash Player to redraw the screen, while setting a style property directly does not always do so.

The StyleManager can also apply styles to all controls using the `global` style name. Global styles apply to every object that does not explicitly override them. This is useful if you want to apply a noninheritable style such as `textDecoration` to many classes at one time.

The following examples illustrate applying the `fontWeight` property to the `ToolTip`, `myStyle`, and `global` style names:

```
// Type selector; applies to all ToolTips.
mx.styles.StyleManager.styles.ToolTip.fontWeight = "bold";

// Class selector; applies to all controls using the style named myStyle.
mx.styles.StyleManager.styles.myStyle.fontWeight = "bold";

// Global style; applies to all controls.
mx.styles.StyleManager.styles.global.fontWeight = "bold";
```

Note: If you set either inheritable and noninheritable styles to the `global` style, Flex applies it to all controls, regardless of their location in the hierarchy.

You can access the values of these properties using the `getStyle()` method or using a reference to the property, as the following examples show:

```
var s1 = mx.styles.StyleManager.styles.global.getStyle("fontWeight");
var s2 = mx.styles.StyleManager.styles.global.fontWeight;
```

The `getStyle()` method requires more computation, so you should use it only when necessary.

The following example defines the `fontFamily` and `fontWeight` style properties for all components using the global style sheet. In addition, it sets the `borderStyle` of all `TextInput` controls to `solid`.

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" width="500"
    height="400" initialize="initializeStyles()">
    <mx:Script><![CDATA[
        import mx.styles.StyleManager;
        function initializeStyles():Void    {
```

```

// Initialize the global settings.
StyleManager.styles.global.fontFamily = "Arial";
StyleManager.styles.global.fontWeight = "bold";

// Initialize all TextInput controls to have solid borders.
StyleManager.styles.TextInput.borderStyle = "solid";
}
]]></mx:Script>
<mx:Button id="btn2" label="Click Me" />
<mx:TextArea width="425" height="250" text="this is a text area" />
</mx:Application>

```

Creating style declaration objects

You can create CSS style declarations using ActionScript with the `mx.styles.CSSStyleDeclaration` class. This lets you create and edit styles at runtime and apply them to classes in your Flex applications. To change the definition of the styles or to apply them during runtime, you use the `setStyle()` method.

You cannot use the `StyleManager` to apply styles to instances of objects, such as an instance of `TextInput`. You can use the `StyleManager` to define and update `CSSStyleDeclaration` objects.

You can use the `setStyle()` method to define style properties in the global style sheet, in a type selector, or in a class selector. The following examples illustrate defining styles in these different scopes:

```

import mx.styles.StyleManager;
function changeStyles():Void {
    // Change the global font to Verdana.
    StyleManager.styles.global.setStyle("fontFamily", "Verdana");

    // Change all TextInput controls to have inset borders.
    StyleManager.styles.TextInput.setStyle("borderStyle", "inset");

    // Change the custom style named "redTahoma18" to be blue.
    StyleManager.styles.redTahoma18.setStyle("color", "0x0000FF");
}

```

Macromedia recommends that you do not use the `setStyle()` method when defining styles during application initialization because it requires more computation than creating a new style and applying it to the component. The following example creates a new style, `redTahoma18`, when the application initializes, and then applies that style using the `StyleManager`:

Note: When you use the `StyleManager`'s `setStyle()` method to set the `color` property, you cannot use the VGA color name for the format. The following example uses the hex `0x` notation.

```

<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" width="500"
height="400" initialize="createStyle()">
<mx:Script><![CDATA[
import mx.styles.StyleManager;
import mx.styles.CSSStyleDeclaration;
var redTahoma18;
function createStyle() {
    // Initialize a custom style named "redTahoma18".
    redTahoma18 = new CSSStyleDeclaration();
}
]]>

```

```

        redTahoma18.color = "0xFF0000";
        redTahoma18.fontFamily = "Tahoma";
        redTahoma18.fontSize = 18;
        // Apply new CSSStyleDeclaration to all TextArea controls.
        StyleManager.styles.TextArea = redTahoma18;
    }
]]></mx:Script>
<mx:TextArea id="ta1" height="200" width="300" text="This is a text area
    used for testing programmatic style application." />
</mx:Application>

```

If you define a style the way the previous example does, you are responsible for all the style properties of the `TextArea` control. In this example, the newly styled `TextArea` controls lack the standard borders, background, and other properties unless you add them to the `redTahoma18` style. If you define styles using CSS, Flex merges the styles together for you.

Using the `setStyle()` and `getStyle()` methods

You interact with the style properties at runtime by using the `getStyle()` and `setStyle()` ActionScript methods. When you use the `getStyle()` and `setStyle()` methods, you can access the style properties of instances of objects or of style sheets. You cannot get or set style properties directly on a component.

Every Flex component exposes these methods. However, the `setStyle()` method is a computationally expensive method to invoke and should only be used when absolutely necessary. You should not use the `setStyle()` method when you are instantiating an object and setting the styles for the first time. It should only be used when you are changing an object's styles during runtime. For more information on improving performance with the `setStyle()` method, see [“Improving performance with `setStyle\(\)`” on page 421](#).

You can also programmatically create and apply style declarations using the `mx.styles.StyleManager` class, which also has `getStyle()` and `setStyle()` methods.

The `getStyle()` method has the following signature:

```
return_type componentInstance.getStyle(property_name)
```

The *property_name* is a `String` indicating the name of the style property (for example, `fontSize`, or `borderStyle`). The *return_type* depends on the style that you access. Styles can be of type `String`, `Number`, or `Boolean`.

The `setStyle()` method has the following signature:

```
componentInstance.setStyle(property_name, property_value)
```

The *property_value* sets the new value of the specified property. To determine valid values for properties, see *Flex ActionScript and MXML API Reference*.

The following example uses the `getStyle()` and `setStyle()` methods to change the `Button`'s `fontSize` style and display the new size in the `TextInput`:

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" width="500"
    height="500">
    <mx:Style>

```

```

    Button {
        fontSize: 10pt;
        color: Blue;
    }
    .myClass {
        fontFamily: Arial, Helvetica, "_sans";
        color: Red;
        fontSize: 22;
        fontWeight: bold;
    }
</mx:Style>
<mx:Script><![CDATA[
    function showStyles() {
        lb1.text=ip1.getStyle("fontSize");
    }
    function setNewStyles(newSize) {
        lb1.text=ip1.setStyle("fontSize",newSize);
    }
]]></mx:Script>
<mx:VBox id="vb">
    <mx:TextInput styleName="myClass" text="My attrs" id="ip1" width="400"/>
    <mx:Label id="lb1" text="" width="400"/>
    <mx:Button label="Show Style" click="showStyles();"/>
    <mx:Button label="Set Style" click="setNewStyles(ip2.text);"/>
    <mx:TextInput text="" id="ip2" width="50"/>
</mx:VBox>
</mx:Application>

```

You can use the `getStyle()` methods to access style properties regardless of how they were set. If you defined a style property as a tag property inline rather than in an `<mx:Style>` tag, you can get this style. However, you cannot override inline style definitions with the `setStyle()` method. You can override style properties that were applied in any other way, such as in an `<mx:Style>` tag or in an external style sheet.

The following example sets a style property inline, and then reads that property with the `getStyle()` method:

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" width="1000"
    height="1000">
    <mx:Script><![CDATA[
        function readStyle() {
            myLabel.text = "Style: " + myLabel.getStyle("fontStyle");
        }
    ]]></mx:Script>
    <mx:VBox xmlns:mx="http://www.macromedia.com/2003/mxml" width="500"
        height="200">
        <mx:Button id="b1" click="readStyle()" label="Get Style" />
        <mx:Label fontStyle="italic" id="myLabel"/>
    </mx:VBox>
</mx:Application>

```

The `setStyle()` method supports only the hexadecimal color format, as the following example shows:

```
btn2.setStyle("color", "0x999933");
```

Improving performance with `setStyle()`

Runtime cascading styles are very powerful, but you should use them sparingly and in the correct context. Dynamically setting styles on an instance of an object means accessing the `UIObject`'s `setStyle()` method. The `setStyle()` method is one of the most resource-intensive calls in the Flex application model framework, because the call requires notifying all the children of the newly styled object to do another style lookup. The resulting tree of children that must be notified can be quite large.

A common mistake that impacts performance is overusing or unnecessarily using the `setStyle()` method. In general, you only need the `setStyle()` method when you want to change styles on existing objects. Do not use it when setting up styles for an object for the first time. Instead, set styles in an `<mx:Style>` block, as explicit style properties on the MXML tag, through an external CSS style sheet, or as global styles. It is important to initialize your objects with the correct style information, if you do not expect these styles to change while your program executes (whether it is your application, a new view in a navigator container, or a dynamically created component).

Some applications need to call the `setStyle()` method during the application or object instantiation. If this is the case, call the `setStyle()` method early in the instantiation phase. Early in the instantiation phase means setting styles from the component or application's `initialize` event, instead of the `creationComplete` or other event. By setting the styles as early as possible during initialization, you avoid unnecessary style notification and lookup.

Using inline styles

You can set style properties as properties of the component in the MXML tag. Inline style definitions take precedence over any other style definitions. The following example defines a type selector for Button components, but then overrides the `color` with an inline definition:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" width="500"
  height="200">
  <mx:Style>
    Button {
      fontSize: 10pt; color: Red;
      fontStyle: italic;
      color: 0x999933;
    }
  </mx:Style>
  <mx:Button label="Inset border"></mx:Button>
  <mx:Button color="0x999933" label="New Color"></mx:Button>
</mx:Application>
```

If you set a style property inline, you cannot override it with any other styles. For example, if you declare a button with a color red (`<mx:Button id="btn1" color="0x999933"/>`), you cannot override it with a `setStyle()` method on the button instance (`btn1.setStyle("color", "0x999933");`).

When setting style properties inline, you must adhere to the ActionScript style property naming syntax rather than the CSS naming syntax. For example, you can set a Button control's `fontSize` property as either `font-size` or `fontSize` in an `<mx:Style>` declaration, but you must set it as `fontSize` in a tag definition:

```
<mx:Style>
    Button { font-size: 15; }
    SimpleButton { fontSize: 15; }
</mx:Style>
<mx:Button fontSize="15" label="My Button"/>
```

When setting color style properties inline, you can use either the hexadecimal format or the VGA color name, as the following example shows:

```
<mx:Button id="btn1" label="Click 1" color="0x9966CC"/>
<mx:Button id="btn2" label="Click 2" color="Yellow"/>
```

About fonts

You define the font that appears in each of your components using the `fontFamily` style property. This property can take a list of fonts, as the following example shows:

```
.myClass {
    fontFamily: Arial, Helvetica;
    color: Red;
    fontSize: 22;
    fontWeight: bold;
}
```

If the client's system does not have the first font in the list, the Flash Player attempts to find the second, and so on, until it finds a font that matches. If no fonts match, Flash Player makes a best guess to determine which font the client uses.

Using device fonts

The safest course when specifying font faces is to include a device font as a default at the end of the font list. *Device fonts* do not export font outline information and are not embedded in the Flash SWF file. Instead, Flash Player uses whatever font on the local computer most closely resembles the device font.

The following example specifies the device font `_sans` to use if the Flash Player cannot find either of the other fonts on the client machine:

```
.myClass {
    fontFamily: Arial, Helvetica, "_sans";
    color: Red;
    fontSize: 22;
    fontWeight: bold;
}
```

Note: You must surround device font names with quotation marks when defining them within style declarations.

Flash includes three device fonts. The following table describes these fonts:

| Font name | Description |
|--------------------------|---|
| <code>_sans</code> | The <code>_sans</code> device font is a sans-serif typeface; for example, Helvetica or Arial. |
| <code>_serif</code> | The <code>_serif</code> device font is a serif typeface; for example, Times Roman. |
| <code>_typewriter</code> | The <code>_typewriter</code> device font is a monospace font; for example, Courier. |

Using device fonts does not impact the size of the SWF file because they are included in the Flash Player. Using them in your applications guarantees that your text appears the same across all platforms. However, using device fonts impacts performance of the application because it requires that the Flash Player interact with the local operating system.

Using embedded fonts

Rather than rely on a client machine to have the fonts you specify, you can embed the font information for a single TrueType font family in your Flex application. This means that the font is always available to the Flash Player when running this application and you do not have to consider the implications of a missing font.

Embedded fonts have the following benefits:

- Client does not need the font to be installed on the local machine.
- Embedded fonts are anti-aliased, which means that their edges are smoothed for easier readability. This is especially apparent when the text size is large.
- Embedded fonts can be transparent.
- Embedded fonts can be rotated.
- Text appears exactly as you expect when using embedded fonts.

Using embedded fonts is not always the best solution, however. Embedded fonts have the following drawbacks:

- You can only embed TrueType fonts.
- Embedded fonts increase the file size of your application, because the document must contain font outlines for the text. This can result in longer download times for your users.
- Embedded fonts decrease legibility of the text at sizes below 10 points. All embedded fonts use anti-aliasing to render the font information on the client screen. As a result, fonts may look fuzzy or illegible at small sizes.
- You can only specify a single `fontFamily` style property when embedding fonts. Flex does not support a list of embedded fonts.

Flex supports the CSS syntax for embedding fonts in Flex applications. You use the `@font-face` “at-rule” to specify the source of the embedded font, and then define the name of the font using the `fontFamily` property. The source can be a local font or one that is accessible using a URL. You use this name in your MXML code to refer to the embedded font.

Note: Check your font licenses before embedding any font files in your Flex application. Double-byte fonts may have licensing restrictions that preclude them from being stored as vector information.

Embedded font syntax

To embed TrueType fonts, you use the following syntax in your style sheet or `<mx:Style>` tag:

```
@font-face {  
    src: [url("location"); | local("name")]  
    fontFamily: reference_name;  
    [descriptor: value;]  
}
```

The `src` attribute specifies the location of the `fontFamily`. You can specify either a `url` or a `local` function. The following table describes these functions:

| src Attribute | Description |
|--------------------|--|
| <code>url</code> | Embeds a TrueType font by location by specifying a valid URI to the font. The URI can be relative (for example, <code>/fontfolder/kbar.ttf</code>) or absolute (for example, <code>http://www.macromedia.com/fonts/akbar.ttf</code>). The URI can also use the file protocol (for example, <code>file:///c:/myfonts/akbar.ttf</code>). |
| <code>local</code> | <p>Embeds a TrueType font by name rather than location. You can embed fonts that are locally accessible by the application server's Java Runtime Environment (JRE). These fonts include the <code>*.ttf</code> files in the <code>jre/lib/fonts</code> folder, fonts that are mapped in the <code>jre/lib/font.properties</code> file, and fonts that are made available to the JRE by the OS.</p> <p>In Windows, TTF files in the <code>/windows/fonts</code> directory (or <code>/winnt/fonts</code>) are available to the <code>local</code> function. On Solaris or Linux, fonts that are registered with a font server such as <code>xfs</code> are available.</p> <p>The font name that you specify is determined by the operating system. In general, you do not include the font file's extension, but this is OS-dependent. Consult your operating system documentation for more information.</p> |

You begin by defining the font using the `@font-face` rule, adding a `src` attribute with either the `url` or `local` function, plus a pointer to the embedded `fontFamily`. The following example defines the `akbar` font using the `url` function:

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

You must specify the `url` or `local` function of the `src` descriptor in the `@font-face` declaration. All other descriptors are optional.

After you define a `@font-face`, you define the new `fontFamily` name, or *alias*, as a type or class selector. The following example sets the `fontFamily` type selector for the `Accordion` controls to use the font defined by the `akbar` alias:

```
Accordion {  
    fontFamily: akbar  
}
```


If the specified embedded font was described as an `@font-face` rule, but there were errors in creating the font in the SWF file, Flex logs a warning and displays the default device `_sans` font in place of the embedded font.

Note: Lists of font families are not supported for embedded fonts. You should specify only one family name when using an embedded font.

Do not mix embedded and non-embedded fonts in the same `fontFamily` descriptor.

Adding multiple faces

Using the `@font-face` declaration embeds a single *face* for the font. A face is the general outline that describes the font's appearance. The result is that each *style* of font must include a new font-face declaration. For example, if you want to use bold and plain versions of the akbar font, you must embed akbar twice; once with the `fontWeight` property set to bold, and once for plain font:

```
<mx:Style>
@font-face {
    src:url("akbar.ttf");
    fontFamily: myfont;
}
@font-face {
    src:url("akbar.ttf");
    fontWeight: bold;
    fontFamily: myfontBold;
}
</mx:Style>
```

By default, Flex includes the entire font definition for each embedded font in the application, so you should limit the number of fonts that you use to reduce the size of the application. You can limit the size of the font definition by defining the character range of the font. For more information, see [“Setting character ranges” on page 426](#).

Identifying embedded fonts

The application object has a property called `embeddedfontlist` that stores the names of all fonts that are embedded in the current Flex application. You do this by using the iterator syntax `var... in application.embeddedfontlist`.

The following example iterates over the application object's `embeddedfontlist` property to get a list of fonts embedded in this application:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" width="425"
height="400">
  <mx:Style>
    @font-face { fontFamily:myfont; src:url("fonts/AGENCYR.TTF"); }
    @font-face { fontFamily:myfont2; src:url("fonts/AGENCYB.TTF"); }
    Application { fontFamily:myfont; fontSize: 18pt }
  </mx:Style>
  <mx:Script><![CDATA[
    function getfontlist() {
      var fl:String="";
      for (var foo in application.embeddedFontList) {
```

```

        fl = fl + "application.embeddedFontList." + foo + " = " +
        application.embeddedFontList[foo] + "\n";
    }
    return fl;
}
]]></mx:Script>
...//process the value fl from getfontlist
</mx:Application>

```

If you have a font name, you can determine if it is embedded using the `isFontEmbedded()` method of the `Application` object. The `isFontEmbedded()` method has the following signature:

```
Application.application.isFontEmbedded(fontFace:String):Boolean
```

To call the `isFontEmbedded()` method, pass in the style name for the font face. It returns `true` if that font is embedded, or `false` if it is not. The following example detects if the `AGENCYR.TTF` font is embedded:

```

<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" width="425"
height="400">
  <mx:Style>
    @font-face { fontFamily:myfont; src:url("fonts/AGENCYR.TTF"); }
    Application { fontFamily:myfont; fontSize: 18pt }
  </mx:Style>
  <mx:Script><![CDATA[
    function detectFontType() {
      var f2 = String(Application.application.isFontEmbedded("myfont"));
      return f2;
    }
  ]]></mx:Script>
  <mx:Button label="Click Me" id="btn" click="f1.text=detectFontType();" />
  <mx:Label styleName="mystyle" id="out" text="Result" />
  <mx:TextArea width="400" height="75" id="f1" text="" />
</mx:Application>

```

Caching embedded font faces

Flex caches a specified number of embedded font faces in memory. The default number of fonts is 20. You can change the number of cached fonts by editing the `flex-config.xml` file. For more information, see [“Editing font settings” on page 809](#).

Setting character ranges

By specifying a range of symbols that compose the face of an embedded font, you reduce the size of an embedded font. Each character in a font must be described; if you remove some of these characters, it reduces the overall size of the description information that Flex must include for each embedded font.

You can set the range of glyphs in the `flex-config.xml` file or in the font-face declaration in each MXML file. You specify individual characters or ranges of characters using the Unicode values for the characters, and you can set multiple ranges for each font declaration.

If you use a character that is outside of the declared range, Flex displays nothing for that character.

For more information on character ranges, see the CSS-2 Fonts specification at www.w3.org/TR/1998/REC-CSS2-19980512/fonts.html#descdef-unicode-range.

Setting ranges in font-face declarations

You can set the range of allowable characters in an MXML file using the `unicode-range` attribute of the `font-face` declaration. The following example embeds the `akbar` font and defines the range of characters for the font in the `<mx:Style>` tag:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:Style>
    @font-face {
      fontFamily: akbar;
      src: local("akbar");
      unicode-range:
        U+0020-U+0040, /* Punctuation, Numbers and Symbols */
        U+0041-U+005A, /* Upper-Case A-Z */
        U+005B-U+0060, /* Punctuation and Symbols */
        U+0061-U+007A, /* Lower-Case a-z */
        U+007B-U+007E; /* Punctuation and Symbols */
    }
    TextArea { fontFamily: akbar; }
  </mx:Style>
  <mx:TextArea text="This is My Text Area" />
</mx:Application>
```

Setting ranges in flex-config.xml

You can specify the language and character range for embedded fonts in the `flex-config.xml` file using the `<language-range>` child tag. This lets you define the range once and use it across multiple `font-face` declarations.

The following example creates an `englishRange` and an `otherRange` named ranges in the `flex-config.xml` file:

```
<font>
  <language-range>
    <lang>englishRange</lang>
    <range>U+0020-U+007E</range>
  </language-range>
  <language-range>
    <lang>otherRange</lang>
    <range>U+00??</range>
  </language-range>
</font>
```

In your MXML file, you point to the defined ranges using the `unicode-range` attribute of the `font-face` declaration, as the following example shows:

```
@font-face {
  fontFamily: Excelsior;
  src: local("akbar");
  unicode-range: englishRange;
}
```

Flex includes a file that lists convenient mappings of the Flash MX 2004 UnicodeTable.xml character ranges for use in the Flex configuration file. The file is located at *flex_app_root*/WEB-INF/flex/flash-unicode-table.xml.

The following example shows the predefined range Latin 1:

```
<language-range>
  <lang>Latin I</lang>
  <range>U+0020,U+00A1-U+00FF,U+2000-U+206F,U+20A0-U+20CF,U+2100-U+2183
</range>
</language-range>
```

To make ranges listed in flash-unicode-table.xml available in your Flex applications, copy the range from this file and add them to the flex-config.xml files.

Embedding double-byte fonts

To use double-byte fonts in Flex, you must specify the Unicode character range in the flex-config.xml file. You then reference the name of that range in your style's `@font-face` declaration.

Flex provides predefined character ranges for common double-byte languages such as Thai, Kanji, Hangul, and Hebrew in the *flex_app_root*/WEB-INF/flex/flash-unicode-table.xml file. This file is not processed by Flex, but is included to provide you with ready definitions for various character ranges. For example, the character range for Thai is listed in the flash-unicode-table.xml file as the following:

```
<language-range>
  <lang>Thai</lang>
  <range>U+0E01-U+0E5B</range>
</language-range>
```

To use this language in your Flex application, copy the character range to the *flex_app_root*/WEB-INF/flex/flex-config.xml file. Add the full definition as a child tag to the `<languages>` tag, as the following example shows:

```
<flex-config>
  <fonts>
    <languages>
      <language-range>
        <lang>thai</lang>
        <range>U+0E01-U+0E5B</range>
      </language-range>
    </languages>
  </fonts>
  ...
</flex-config>
```

You can change the value of the `<lang>` element to anything you want. When you embed the font using CSS, you refer to the language using this value in the `unicode-range` property of the `@font-face` declaration, as the following example shows:

```
@font-face {
  font-family:"Thai_font";
  src: url("THAN.TTF"); /* Embed from file */
  unicode-range:"thai"
}
```

CHAPTER 17

Using Themes and Skins

This chapter describes how to use themes and reskin Macromedia Flex components using either ActionScript class files or Flash files.

Contents

| | |
|---|-----|
| About skinning | 429 |
| About themes..... | 432 |
| Programmatically reskinning the CheckBox control..... | 433 |
| Programmatic skinning concepts | 438 |
| Graphical skinning..... | 449 |

About skinning

Skinning is the process of changing the appearance of a component by modifying or replacing its graphical elements. These graphical elements can be made up of images or the output of drawing API methods. They are known as *symbols*. You can reskin Flex components without changing their functionality. A file that contains new skins for use in your Flex applications is known as a *theme*.

There are two types of skins in Flex: graphical and programmatic. *Graphical skins* are Macromedia Flash symbols that you can change directly in the Flash MX 2004 authoring environment. You draw *programmatic skins* using ActionScript statements and define these skins in class files. Sometimes it is more advantageous to reskin a component graphically, and in some cases it makes more sense to reskin a component programmatically. You cannot combine graphical and programmatic skins in a single theme file.

To change the appearance of controls that use programmatic skins, you edit an ActionScript class file, compile that file as a SWC file, and then deploy that SWC file as a theme with your Flex application.

To change the appearance of controls that use graphical skins, you edit the symbols in the Flash MX 2004 environment, and export the resulting FLA file as a SWC, and then deploy that SWC file with your Flex application.

One advantage of programmatic skinning is that it provides you with a great deal of control over styles. For example, you cannot control the radius of a Button control's corners with style sheets without reskinning the component. You can also develop programmatic skins can also be developed directly in your Flex authoring environment or any text editor, without using a graphic tool such as Flash.

If you do not add styles to skins, graphical skins may be easier to create. Graphical skins are edited in Flash and, just like programmatic skins, they are exported to a SWC file.

When reskinning your application, you should only create skins for the controls that are changing appearance. For example, if you are only changing the appearance of Accordion headers, your theme file should only contain a new master skin symbol for Accordion. Flex uses the default skins for any skins that are not explicitly overwritten. Your application automatically applies any future changes to the default skins.

You can also reskin Flex components by creating a custom component in Flash and changing the properties of the skins in the component's class files. This technique is recommended for only experienced Flash designers.

Skinning resources

Flex includes sample files for both programmatic and graphical skins. You can use these samples as a starting point from which you can change the appearance of most Flex components. Not all components have programmatic skin files. In some cases, you can only reskin a component graphically.

Programmatic skin samples are defined by ActionScript class files. To programmatically reskin Flex components, you edit these files. For more information, see [“Programmatic skinning concepts” on page 438](#).

Graphical skins are available in the `pulseBlue fla` and `pulseOrange fla` theme files. To graphically reskin Flex components, you edit these files in the Flash MX 2004 environment. The difference between the `pulseBlue fla` and `pulseOrange fla` files is the set of hues used to color the component's various states. For more information, see [“Graphical skinning” on page 449](#).

About the master skin symbol

Every component has one master skin symbol. This is the single linkage between a component and all of its skin elements. For example, the Button control has a single master skin symbol called `ButtonSkin`.

Some components are made up of multiple skins, but they still have a single master skin symbol. For example, the `ScrollBar` master skin symbol is `ScrollBarAssets`. The other scroll bar skin files (such as `ScrollUpArrow` and `ScrollDownArrow`) are not master skin symbols; they are individual skin elements referenced by the master skin symbol.

For graphical skins, the master skin symbol is the “Identifier” name in the Linkage dialog box, which can be different from the Library symbol name. For programmatic skins, the master skin symbol is the `symbolName` class property.

When reskinning Flex components, you create a new theme file that replaces the existing master skin symbol with a new graphical or programmatic definition. You are removing the default skin and replacing it with your skin. You are not adding to the overall size of the application.

The following table lists Flex controls and their master skin symbols. In addition, it specifies the programmatic skin files, if any, that you edit to reskin this symbol. All the components listed in this table can be graphically reskinned using the Pulse theme files described in [“Skinning resources” on page 430](#). Some controls have multiple master skin symbol files, so to change the appearance of that control, you might be required to edit more than one ActionScript file.

| Control | Master skin symbol | Programmatic skin file |
|----------------|---------------------------------|--|
| Accordion | AccordionHeaderSkin | SampleAccordionHeaderSkin.as |
| Button | ButtonSkin | SampleButtonSkin.as |
| CheckBox | CheckBoxAssets | SampleCheckBoxIcon.as |
| ComboBox | ComboAssets | SampleComboBoxArrowSkin.as SampleRectBorder.as |
| DataGrid | DataGridAssets | None |
| DateChooser | DateChooserAssets | None |
| DateField | DateFieldAssets | None |
| DividedBox | mx.skins.BoxDividerSkin | None |
| HDividedBox | mx.skins.cursor.HBoxDivider | |
| VDividedBox | mx.skins.cursor.VBoxDivider | |
| FormItem | mx.containers.FormItem.Required | None |
| Menu | MenuAssets | None |
| MenuBar | MenuBarAssets | None |
| NumericStepper | NumericStepperAssets | SampleNumericStepperAssets.as SampleNumericStepperDownSkin.as SampleNumericStepperUpSkin.as |
| Panel | WindowAssets | SampleTitleBackground.as |
| TitleWindow | | |
| Alert | | |
| ProgressBar | ProgressBarAssets | SampleProgressBarAssets.as SampleProgressBarSkin.as SampleProgressIndeterminateSkin.as SampleProgressTrackSkin.as |
| RadioButton | RadioButtonAssets | SampleRadioButtonIcon.as |
| ScrollBar | ScrollBarAssets | SampleScrollBarAssets.as |
| HScrollBar | | SampleScrollArrowDownSkin.as |
| VScrollBar | | SampleScrollArrowUpSkin.as |
| | | SampleScrollThumbSkin.as SampleScrollTrackSkin.as |

| Control | Master skin symbol | Programmatic skin file |
|-------------------|--------------------|------------------------|
| Slider HSlider | HSliderAssets | None |
| VSlider | VSliderAssets | None |
| Tab | mx.skins.TabSkins | SampleTabSkin.as |
| Tree | TreeAssets | None |

About themes

A *theme* defines the look of a Flex application. It is a collection of styles and skins that make up a component's appearance. The theme of an application can manifest itself as a color scheme or a distinctive brush that you use to draw icons and other onscreen elements. A theme can also add a subtle change, such as a light shadow, that you add to existing graphics to make the application's appearance distinctive. You can only apply one theme file to each application.

When you reskin a Flex component, you create a new theme file for your Flex application. In Flex, themes take the form of a SWC file. You compile this file out of programmatic skin files or from the Flash authoring environment.

Themes define what styles are available for you to set in your Flex applications. If the theme does not define styles such as `fillColor`, you cannot set the `fillColor` style property in your application. The default theme included with Flex supports a subset of the style properties. For more information, see [“About supported styles” on page 410](#).

When reskinning programmatically, you can define to which properties you can apply styles. For more information, see [“Making skin properties styleable” on page 447](#).

Using the theme property

You apply a single theme to your Flex application using the `theme` property of the `<mx:Application>` tag. The following example applies the `FreakyStyley` theme to the application:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" width="500"
    height="300" theme="../../themes/FreakyStyley.swc">
```

You can give the theme SWC file any name. In the `<mx:Application>` tag, set the value of the `theme` property to the name of the SWC file.

Flex treats theme SWC files differently than other SWC files. You must store the custom theme's SWC file in a separate directory from the application files and from other SWC files. Flex searches for the specified SWC file in a location relative to the MXML file.

Do not store the theme file in the same directory as the application root or in a directory specified by the `<lib-path>` settings in `flex-config.xml`. If you do, Flex can produce unexpected results.

About theme limitations

Applying a new theme to a Flex application is designed to be simple. However, because of its simplicity, you should be aware of the following limitations:

- You can assign only one SWC file as the theme for an application. If you want to use skins from multiple Flash or ActionScript files, you must combine them into a single SWC file.
- You cannot combine both programmatic skins and graphical skins in a single SWC file.
- You cannot apply skins to a single control or an instance of a control. The symbols in the SWC file are applied to all components in your application. If you reskin the Button control, for example, all Button controls in your application use the new skin.
- Components are sometimes composed of more than one skin. You might be required to edit multiple files in order to reskin a single component. For example, to programmatically reskin the NumericStepper component, you might be required to edit the SampleNumericStepperAssets.as, SampleNumericStepperDownSkin.as, and SampleNumericStepperUpSkin.as files.
- Some components share the same skins. As a result, components that share skins may use the same skins for different purposes. When you apply a new theme, you must be aware that the theme can unintentionally apply to some components.
- Symbols in the theme SWC file take precedence over all other symbols in the component definitions. If the SWC file includes symbols that you do not want to use in your Flex application, you must remove those symbols.
- Class implementations found in the application or its libraries generally take precedence. Do not attach graphic resources directly to class definitions that have the same name as library classes, because Flex ignores those assets.

Programmatically reskinning the CheckBox control

The CheckBox control includes an icon (the box), which can have a check mark in it, and a label. You can change the appearance of the check box and the check mark inside the box by using programmatic skinning. You cannot reskin the CheckBox control's label. It is not a skinnable asset; it is instead defined by style properties.

This section introduces you to the concepts involved in programmatically reskinning a Flex component. In this example, you will reskin the CheckBox control by following these steps:

- [“Prepare the skin file for custom skinning” on page 434](#)
- [“Change the color of the check mark” on page 434](#)
- [“Make the check mark color styleable” on page 435](#)
- [“Change the size of the check box” on page 436](#)
- [“Compile the skin file” on page 436](#)
- [“Deploy the new skin” on page 437](#)

This section guides you through the process of reskinning one component. For more information on the concepts in this section, see [“Programmatic skinning concepts” on page 438](#).

Prepare the skin file for custom skinning

Flex includes a sample ActionScript class file that defines the programmatic skin of the CheckBox component. Find the `SampleCheckBoxIcon.as` file in the *flex_install_dir/resources/themes/*programmatic directory, and save it to a new location. You can optionally give it a new filename; for example, save it as `MyCheckBoxIcon.as` in your `c:\custom_skins` directory.

At the top of the ActionScript file, change the class name and `symbolOwner` property to match the new name of the file. Ensure that you also change the class's package if necessary. By default, the sample classes are located in the `sampletheme` package.

The `symbolName` property should be the name of the master skin symbol. In this case, `CheckBoxAssets`. You should not have to change the value of this property.

The following example renames the class and `symbolOwner` property:

```
class MyCheckBoxIcon extends RectBorder {
    static var symbolName:String = "CheckBoxAssets";
    static var symbolOwner:Object = MyCheckBoxIcon;
    ...
}
```

Change the color of the check mark

Part of reskinning the CheckBox control is changing the color of the check mark in the check box. You can change the color of the check mark for each state. In this example, you change the default color of the check mark to blue (0x3399FF), and then change the color of the disabled check mark to gray (0xCCCCCC).

```
switch (borderStyle) {
    case "truedisabled":
        checkColor = 0xCCCCCC; //was 0xAAAAAA
        bDrawCheck = true;
        break;
    case "trueup":
        checkColor = 0x3399FF; // was 0xEEEEEE
        bDrawCheck = true;
        break;
}
```

The switch statement uses evaluates the `borderStyle` property and executes the appropriate case statement. This property stores the current state as a string value (for example, `trueup`). All programmatic skins that have multiple states use the `borderStyle` property to determine the current state. Some skins, such as `ScrollBarTrack` and `ProgressBar`, have a single state and do not use the `borderStyle` property.

After a change in the color of the check mark, you must redraw the check mark so that Flex displays the new color. The CheckBox skin file uses the `bDrawCheck` variable to determine when to redraw the check mark. Its initial value is `false`, which means that Flex will not, by default, redraw the check mark. After the switch statement is a check against the value of `bDrawCheck`. If it is `true`, Flex redraws the check mark, as the following example shows:

```
if (bDrawCheck) {
    lineStyle(2,checkColor,100);
}
```

```

        moveTo(3, 3);
        lineTo(w-3, h-3);
        moveTo(w-3, 3);
        lineTo(3, h-3);
    }

```

Make the check mark color styleable

You can make the `checkColor` skin property a styleable property. This means that you can change the color of the check mark at runtime using the `setStyle()` method or CSS properties in your Flex application.

To make the `checkColor` property styleable, call the `getStyle()` method on it toward the beginning of the `drawIcon()` method.

Instead of using a constant value for `checkColor`, you now call `getStyle()` to retrieve the color value dynamically, as the following example shows:

```

function drawIcon(w:Number,h:Number):Void {
    var borderStyle = getStyle('borderStyle');
    var fillColor;
    var borderColor = 0x999999;
    var bDrawCheck = false;
    //var checkColor = 0x3399FF;
    var checkColor = getStyle('checkColor');
    ...
}

```

In your Flex application, you now set the default color of the check mark in either a CSS style or in a `setStyle()` method. The following example uses CSS to set the color of the check mark:

```

<mx:Style>
    CheckBox {
        checkColor:#3399FF;
    }
</mx:Style>

```

The following example uses the `setStyle()` method to set the color of the check mark when the application initializes:

```

<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    initialize="setStyles();">
    <mx:Script>
        function setStyles() {
            checkbox1.setStyle("checkColor",0x3399FF);
        }
    </mx:Script>

    <mx:Panel title="Skin Tester">
        <mx:CheckBox label="This is a CheckBox" id="checkbox1" />
    </mx:Panel>
</mx:Application>

```

Change the size of the check box

You can change the size of the check box in the `CheckBox` control. You do this in the skin class's `init()` function by changing the value of the `layoutWidth` and `layoutHeight` properties. The following example increases the size of the check box to 19 pixels x 19 pixels:

```
function init():Void {  
    super.init();  
    layoutWidth = __width = _preferredWidth = 19; // was 12  
    layoutHeight = __height = _preferredHeight = 19; // was 12  
}
```

When you change the size of the check box, Flex recalculates the length of the lines that make up the check mark. No additional changes are necessary to make the check mark itself bigger.

You can also change the border around the box by editing the line style that is used to draw that border. To make the border bigger, change the value of the first parameter in the `lineStyle()` method to a number greater than 1.

The following code draws the check box with a thicker line:

```
lineStyle(2, borderColor, 100); // Size was 1  
beginFill(fillColor, 100);  
moveTo(0, 0);  
lineTo(w-1, 0);  
lineTo(w-1, h-1);  
lineTo(0, h-1);  
lineTo(0, 0);  
endFill();
```

This section of code that redraws the check box is after the switch statement.

You can change the gap between the icon and label by using the `horizontalGap` style property, and the style of the text by using the text style properties (such as `fontSize` and `fontFamily`). Setting these properties do not require custom skinning. You cannot change the vertical positioning of the text.

Compile the skin file

When you finish editing the sample skin class, you compile the new skin into a SWC file. To do this, you use the `compc` utility.

When using the `compc` utility, you specify a `-root` and `-o` option. The `-root` option specifies the location of the Flex application that will use the SWC file. The `-o` option specifies the output location and name of the new SWC file.

In this example, you create the `myTheme.swc` file in the current working directory:

```
compc -root c:\Jrun4\servers\flex\myApp MyCheckBoxIcon.as -o myTheme.swc
```

The `compc` utility creates the theme SWC file that you deploy.

Deploy the new skin

After you compile the new SWC file, you use it as a theme file in your application. Copy the SWC file to a location that is accessible to the Flex application, but not in the root directory of the application.

To deploy the new skin:

1. Create a /themes subdirectory in your Flex application. You can create a new directory with any name, but you must ensure that you avoid putting theme files in the application's root directory.

Note: Do not store the theme file in the *flex_app_root*/WEB-INF/flex/user_classes directory or any directory specified by the `<lib-path>` child tag in the *flex-config.xml* file. Putting the theme's SWC file in one of these directories can cause versioning errors.

2. Copy the *myTheme.swc* file to the new directory.
3. Create a new application that uses a *CheckBox* control. In the `<mx:Application>` tag, set the value of *theme* attribute to point to the new theme file.

The following sample application includes *ActionScript* functions that let you change the color of the check mark and enable and disable the controls:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    theme="mytheme/mytheme.swc">
    <mx:Script>
        function enableControls(enabled:Boolean):Void {
            button1.enabled = enabled;
            button2.enabled = enabled;
            checkbox1.enabled = enabled;
        }

        function changeColor(c) {
            checkbox1.setStyle("checkColor",c);
        }
    </mx:Script>

    <mx:Panel title="Test the New CheckBox Skin">
        <mx:Button label="Change Color" id="b1" click="changeCheck(0xFF99FF)"/>
        <mx:Button label="Revert" id="b2" click="click=changeCheck(0x3399FF)"/>
        <mx:CheckBox label="CheckBox" id="checkbox1" />
        <mx:ControlBar>
            <mx:Button label="Disable" click="enableControls(false)" />
            <mx:Button label="Enable" click="enableControls(true)" />
        </mx:ControlBar>
    </mx:Panel>
</mx:Application>
```

Programmatic skinning concepts

To programmatically reskin a component, you edit the ActionScript skin files for the component. These files usually contain calls to drawing API methods, as well as styling of the component. You then compile a SWC file from this file, and reference that new SWC file as a theme in the Flex application.

This section describes how to reskin components programmatically.

Editing programmatic skin files

The ActionScript files for programmatic skinning contain references to all the necessary skin classes, styling information, and drawing calls that draw the component's skin.

Most components have multiple states, and these states are usually defined within a switch/case statement. As a result, to reskin a component entirely, you might be required to change several properties in multiple states.

The ActionScript file defines the appearance of all states of a control. For example, a CheckBox control has eight states, representing the appearance of the component when it is enabled, disabled, moused over, and others. The following are the states for the CheckBox component:

- falseup
- falsedown
- falserollover
- falsedisabled
- trueup
- truedown
- truerollover
- truedisabled

You can edit the appearance of the control in any number of its states, but if your application does not allow controls to be disabled, you can ignore the disabled states. In addition, you might comment out all the code in the rollover states if you do not want your control to behave differently when the user moves their mouse pointer over the control. Limiting the interactivity of a component is one form of reskinning.

To reskin Flex controls programmatically, you use the following general process:

1. Copy the sample ActionScript class file into your working skin development directory.
2. Follow the directions in the class file to update the filename and class name.
3. Debug your new skin file.
4. Compile the skin into a theme SWC file.
5. Deploy the skin by adding a `theme` attribute to your Flex application. This attribute points to the new theme SWC file.

The following sections describe each of these steps in detail.

General guidelines

Each component skin file is different, however, there are some general guidelines that you can follow when you edit the sample ActionScript skin files that are included with Flex. This section describes some general guidelines which you can use if you create your own ActionScript skin files.

Importing classes

Ensure that you import all the necessary files in the programmatic skin classes. Most programmatic skins import the `mx.core.ext.UIObjectExtensions` and the `mx.skins.RectBorder` classes. Also, programmatic skin classes must import the component's class.

The following example shows the classes that the `SampleRadioButtonSkin.as` file imports:

```
import mx.controls.RadioButton;
import mx.core.ext.UIObjectExtensions;
import mx.skins.RectBorder;
```

The `RectBorder` class is the base class for all ActionScript skin files. For more information, see [“About RectBorder” on page 443](#).

The `SampleProgressBarAssets.as` and `SampleNumericStepperAssets.as` classes do not extend `RectBorder`. Instead, they extend `MovieClip`. You do not typically edit these files because they have associated skin files (`SampleProgressBarSkin.as`, `SampleNumericStepperDownSkin.as` and `SampleNumericStepperUpSkin.as`) that you would edit.

The sample skin files included with Flex import the appropriate classes. You should not have to change or add to the import statements when editing these sample files.

Defining symbols

All programmatic skin files must define the `symbolName` and `symbolOwner` properties. The `symbolName` property is the master skin symbol name for the skin. The `symbolOwner` property is the name of the programmatic skins file's ActionScript class. This property matches both the class name and the file name.

The following example defines the `symbolName` and `symbolOwner` for a new `CheckBox` skin in the `MyCheckBoxIcon.as` file:

```
static var symbolName:String = "CheckBoxAssets";
static var symbolOwner:Object = MyCheckBoxIcon;
```

The sample skin files included with Flex already define these properties. If you change the name of the files (for example, from `SampleProgressBarSkin.as` to `MyProgressBarSkin.as`), you must also change the class name and the `symbolOwner` property.

You can only change the `symbolName` if it is the same as the class name. The `symbolName` property must match the master skin symbol name only for master skin symbols. Other skins, such as `ScrollUpArrow`, can have an arbitrary `symbolName`. If you do change the `symbolName` property, you must perform a global search and replace for that property. The `symbolName` property is also used in the `classConstruct()` method.

Initialization

Because most programmatic skins extend the `mx.skins.RectBorder` interface, they often override the `init()` method. This occurs when Flex sets a static size for the component or the component icon.

If you do override this method, you must at least call `super.init()`, as the following example shows:

```
function init():Void {
    super.init();
    layoutWidth = __width = _preferredWidth = 16;
    layoutHeight = __height = _preferredHeight = 1;
}
```

You must override the `init()` method if you explicitly define the `layoutWidth` and `layoutHeight` property of the component. In some cases, you cannot set the height and width of the component. For example, if you reskin the Button control, Flex determines the size of the Button control after applying the skin. As a result, setting the height and width in the `init()` method or any other method has no effect.

About drawing

The code that draws component skins is in the `drawSkin()` or `drawIcon()` methods of the ActionScript skin files. The latter method applies only to components that have icons (CheckBox and RadioButton).

Inside these methods, Flex determines the state, draws the appropriate graphics, and sets the appropriate styles.

The `drawSkin()` and `drawIcon()` methods take two arguments: `h` and `w`. These arguments correspond to the height and width of the area on which Flex draws the component's skin. In many of the drawing API calls, the programmatic skins use these measurements to determine the size and length of the lines and fills.

For more information on the drawing API, see [“Drawing programmatically” on page 444](#).

Determining state

When Flex draws a component on the screen in the `drawSkin()` or `drawIcon()` methods, it determines what state the component is in, and then displays the appropriate skin for that state. For example, when the user clicks on a Button control, Flex displays the skin that corresponds to the Button control's `true`down state.

To determine state, you access the value of the `borderStyle` property. This is the same for all programmatically skinned components. This property contains a string that defines the state (such as `true`down or `false`down).

The following example shows the case statement for the `CheckBoxIcon` skin file. This shows you the eight possible states for this control. In addition, it shows you that some states have different colors, and others require no changes:

```
var themeColor = getStyle("themeColor");
var state = getStyle("borderStyle");
```



```

switch (state) {
    case 'falseup':
        break;
    case 'falsesrollover':
        borderColor = themeColor;
        bgColor = 0xEEEEEE;
        break;
    case 'falsedown':
        borderColor = themeColor;
        bgColor = themeColor225;
        break;
    case 'falsedisabled':
        borderColor = 0xAAAAAA;
        bgColor = 0xEEEEEE;
        break;
    case 'trueup':
        bDrawCheck = true;
        break;
    case 'truerollover':
        bDrawCheck = true;
        borderColor = themeColor;
        bgColor = 0xEEEEEE;
        break;
    case 'truedown':
        bDrawCheck = true;
        borderColor = themeColor;
        checkColor = 0x999999;
        bgColor = themeColor;
        break;
    case 'truedisabled':
        bDrawCheck = true;
        borderColor = 0xAAAAAA;
        bgColor = 0xEEEEEE;
        checkColor = 0xAAAAAA;
        break;
}

```

Skin files typically use the drawing API to redraw the skin either in the case statements or after the switch block. For example, in the case of the `SampleButtonSkin.as` file, the drawing methods are inside each case statement.

Calling the `clear()` method

When a component undergoes a change such as displaying a new state or drawing a new graphic, Flex must remove the existing skin's graphics so that there is no bleeding or overlapping of images. To do this, Flex uses the `clear()` method.

The `clear()` method removes all the graphics that were created during runtime using the `MovieClip` draw methods. It also resets any line style that was specified with the `lineStyle()` method. In general, you should call the `clear()` method before any drawing methods such as `lineTo()`. This ensures that the area is clear before adding the component's shapes.

In most cases, the programmatic skins call the `clear()` method directly before the `switch` statement because the component is redrawn inside the various cases within that statement or at the end of the statement:

```
clear();
switch (borderStyle) {
    case "falseup":
        lineTo(...)
    }
    case "falsedown":
        lineTo(...)
}
```

The `clear()` method does not reset style properties or the values of other variables in the programmatic skin classes.

Sizing static elements

Skins with static dimensions (such as icons or simple buttons) declare their dimensions inside the `init()` method. For example, to change the size of the box in a `CheckBox` component, change the value of the `layoutWidth` and `layoutHeight` properties in the `init()` method, as the following example shows:

```
function init():Void {
    super.init();
    layoutWidth = __width = _preferredWidth = 18; //was 12
    layoutHeight = __height = _preferredHeight = 18; //was 12
}
```

The following controls have sample skin files that include static sizing of elements:

- `CheckBox`
- `ComboBox`
- `NumericStepper`
- `ProgressBar`
- `RadioButton`
- `ScrollBar`

Sometimes, you cannot statically size a component because Flex dynamically sizes them after the skin has been applied. Setting the height or width of the component in this case has no effect.

About the `classConstruct()` method

Each sample skin file has a `classConstruct()` method at the bottom of the file in addition to other declarations.

The `classConstruct()` method is used internally by Flex when applying the skins to the components. In most cases, you should not edit any code inside this method, nor should you add or remove any of the code. In addition, do not change the values of the “dependency” variables at the end of the file.

If you change the `symbolName` of the master skin symbol, you must edit the contents of the `classConstruct()` method.

About RectBorder

RectBorder defines the programmatic skin for all component borders. This includes the basic border types `solid`, `inset`, `outset`, and `none`, as well as custom border types like `dropDown` and `controlBar`.

Most programmatic skins extend the RectBorder class, which means that changes to this class affect the borders of most controls that have programmatic skins. The contents of this class define the outline and background fill of the component. You can create a custom border by editing the sample programmatic skin `SampleRectBorder.as`.

In the `SampleRectBorder.as` file you can change:

- Border thickness
- Color and fill properties of the borders
- Shape and size of the border

You can also create entirely new border styles by adding them to the `SampleRectBorder.as` file.

You can change the size of a basic border types by adding case statements in the `getBorderMetrics()` method of this class. For example, to radically increase the size of the `inset` border type, add the following `inset` entry:

```
function getBorderMetrics():Object {
    ...
    case "inset":
        __borderMetrics = {left:10, top:5, right:10, bottom:5};
        break;
    ...
}
```

Then apply the new style with a class or type selector:

```
<mx:Style>
    .myStyle {
        borderStyle: inset;
    }
</mx:Style>

<mx:VBox height="100" width="200" styleName="myStyle">
    ...
</mx:VBox>
```

You cannot set custom style properties (including new values for existing styles like `borderStyle`) as attributes on a component. Instead, you must set them with a class or type selector in CSS.

You compile the `SampleRectBorder.as` file as you would any other programmatic skin file. However, when you deploy a theme that includes a reskinned `SampleRectBorder.as` file, Flex applies the new border styles to all components that use them.

In the `drawBorder()` method, you edit the fills, fill colors, and lines that define the borders. For example, you can add the `inset` case statement and then define the color of the border for that `borderStyle`, as the following example shows:

```
function drawBorder() {
    var borderStyle = getStyle('borderStyle');
    switch (borderStyle) {
        case "dropDown":
            ...
        case "controlBar":
            ...
        case "inset":
            var borderColor = 0x00FF66;
            break;
        default:
            ...
    }
}
```

If you want to add a fill, copy the definitions for the `dropDown` or `controlBar` styles and modify them to fit your needs. The following example, based on the `controlBar` style, adds a radial fill to the custom `borderStyle`:

```
case "myNewStyle":
    var fillColors = [0x000000,0xAAAAAA];
    var alphas = [100, 100];
    var ratios = [0, 255];
    var matrix = {matrixType:"box", x:0, y:0, w:layoutWidth,
        h:layoutHeight, r:0};
    clear();
    // Draw the gradient background.
    beginGradientFill("radial", fillColors, alphas, ratios, matrix);
    ...
```

In the `RectBorder` class, the `borderStyle` property has a different meaning than it does in the other sample skin files. It is actually the style of the border, which is defined by `RectBorder` (for example, `inset` and `outset`); in the other sample programmatic `ActionScript` skin classes, `borderStyle` refers to the state of the component (for example, `falseup` and `falsedown`).

Drawing programmatically

You use the drawing methods of the `MovieClip` object to draw the parts of a programmatic skin. These methods let you describe fills or gradient fills, define line sizes and shapes, and draw lines. By combining these very simple drawing methods, you can create complex shapes that make up your component skins.

The following table briefly describes the drawing API methods:

| Method | Summary |
|----------------------------------|---|
| <code>beginFill()</code> | <p>Begins drawing a fill. The <code>beginFill()</code> method has the following signature:</p> <pre>beginFill([rgb[, alpha]])</pre> <p>For example:</p> <pre>beginFill(0xCCCC,100);</pre> <p>If an open path exists (that is, if the current drawing position does not equal the previous position that you specified in a <code>moveTo()</code> method) and it has a fill associated with it, that path is closed with a line and then filled.</p> |
| <code>beginGradientFill()</code> | <p>Begins drawing a gradient fill. The <code>beginGradientFill()</code> method has the following signature:</p> <pre>beginGradientFill(fillType, colors, alphas, ratios, matrix)</pre> <p>For example:</p> <pre>colors = [0xFF0000, 0x0000FF]; alphas = [100, 100]; ratios = [0, 0xFF]; matrix = { a:200, b:0, c:0, d:0, e:200, f:0, g:200, h:200, i:1 }; beginGradientFill("linear", colors, alphas, ratios, matrix);</pre> <p>If an open path exists (that is, if the current drawing position does not equal the previous position that you specified in a <code>moveTo()</code> method), and it has a fill associated with it, that path is closed with a line and then filled.</p> |
| <code>clear()</code> | <p>Removes all the drawing output associated with the current <code>MovieClip</code> object. The <code>clear()</code> method takes no arguments.</p> |
| <code>curveTo()</code> | <p>Draws a curve using the current line style. The <code>curveTo()</code> method has the following signature:</p> <pre>curveTo(controlX, controlY, anchorX, anchorY)</pre> <p>For example:</p> <pre>moveTo(500, 500); curveTo(600, 500, 600, 400); curveTo(600, 300, 500, 300); curveTo(400, 300, 400, 400); curveTo(400, 500, 500, 500);</pre> |
| <code>endFill()</code> | <p>Ends the fill specified by <code>beginFill()</code> or <code>beginGradientFill()</code> method. The <code>endFill()</code> method takes no parameters. If the current drawing position does not equal the previous position that you specified in a <code>moveTo()</code> method and a fill is defined, the path is closed with a line and then filled.</p> |

| Method | Summary |
|--------------------------|---|
| <code>lineStyle()</code> | <p>Defines the stroke of lines created with subsequent calls to the <code>lineTo()</code> and <code>curveTo()</code> methods. The <code>lineStyle()</code> method has the following signature:</p> <pre>lineStyle([thickness[, rgb[, alpha]]])</pre> <p>The following example sets the line style to a 2-point gray line with 100% opacity:</p> <pre>lineStyle(2,0xCCCCCC,100)</pre> <p>You can call the <code>lineStyle()</code> method in the middle of drawing a path to specify different styles for different line segments within a path. Calls to the <code>clear()</code> method reset line styles back to undefined.</p> |
| <code>lineTo()</code> | <p>Draws a line using the current line style. The <code>lineTo()</code> method has the following signature:</p> <pre>lineTo(x,y)</pre> <p>The following example draws a triangle:</p> <pre>moveTo (200, 200); lineTo (300, 300); lineTo (100, 300); lineTo (200, 200);</pre> <p>If you call <code>lineTo()</code> before any calls to the <code>moveTo()</code> method, the current drawing position defaults to (0, 0).</p> |
| <code>moveTo()</code> | <p>Moves the current drawing position to the specified coordinates. The <code>moveTo()</code> method has the following signature:</p> <pre>moveTo(x,y)</pre> <p>For example:</p> <pre>moveTo(100,10);</pre> |

The following example from the `CheckBox` programmatic skin defines a line and a fill, draws a box within a box, and fills it:

```
lineStyle(2, borderColor, 100);
beginFill(fillColor, 100);
moveTo(0, 0);
lineTo(w-1, 0);
lineTo(w-1, h-1);
lineTo(0, h-1);
lineTo(0, 0);
endFill();
```

Also from the `CheckBox` skin file, the following example creates a new line style and draws a check mark in the box that was created in the previous example:

```
lineStyle(2,checkColor,100);
moveTo(3, 3);
lineTo(w-3, h-3);
moveTo(w-3, 3);
lineTo(3, h-3);
```

You can use any method available to the `MovieClip` object to draw skins. For more information, see *Flex ActionScript Language Reference*.

In many cases, the programmatic skin file uses the `h` (height) and `w` (width) properties to determine positioning and sizing when drawing. These values are passed into the `drawIcon()` method and are equivalent to the `layoutHeight` and `layoutWidth` properties and sometimes are set in the `init()` method.

In some cases, the values of `h` and `w` are not the entire height and width of the component, but rather the height and width of an icon that Flex draws. The `CheckBox` and `RadioButton` controls have labels that are outside of the area of their icons. The labels are not included when determining the height and width for `h` and `w`. In all other programmatic skins, such as `ButtonSkin` and `TabSkin`, `h` and `w` are the size of the entire component.

Making skin properties styleable

When you programmatically reskin a component, you can specify which style properties, if any, your skin responds to. This lets you set the value of the style property in your Flex application's ActionScript code blocks or in a style sheet. Making a new skin property styleable gives you more flexibility when defining the component's appearance. You can also dynamically set a default value when the component is first initialized.

It is important to note that you cannot embed style sheets into a skin's theme SWC file. If you use style sheets with your theme, you must deploy them separately to your Flex application.

To make a custom skin setting styleable, add a `getStyle()` method inside the skin file's draw methods. For example, in the `SampleCheckBoxIcon.as` ActionScript file, add the following method inside the `drawIcon()` method to make the `checkColor` property styleable:

```
var checkColor = getStyle('checkColor');
```

In your Flex application, you can then change the value of the `checkColor` property with a call to the `setStyle()` method, as the following example shows:

```
<mx:Button label="Change Color"
  click="myCheckBox.setStyle('checkColor',0x3399FF);" />
```

If you do not add a `getStyle()` method for a skin setting, you cannot set the value of that property with CSS or a `setStyle()` method.

Adding a `getStyle()` method to the `drawIcon()` method can cause the default setting to be overwritten by an undefined value; in other words, the value of the property is set to `undefined` and not defined as the default color during instantiation. As a result, you should set the initial value of a styleable skin property in your Flex application so that your component appears as you intended. Do this during initialization or using a CSS style setting, such as the following:

```
<mx:Style>
  CheckBox {
    checkColor:#3399FF;
  }
</mx:Style>
```

Custom skin properties that you define as styleable are noninheritable. As a result, subclasses or children of that component do not inherit the value of that property.

Compiling programmatic skins

After you edit a programmatic skin's ActionScript file, you compile and deploy it with your Flex application as a theme file.

You compile the skin files into a SWC file by using the `compc` utility. The following example creates the `myTheme.swc` file from the `MyCheckBoxIcon.as` file with `compc`:

```
compc -root c:\JRun4\servers\flex_mainline\flex\ -o myTheme.swc
MyCheckBoxIcon.as
```

You can combine multiple ActionScript skins into a single SWC file using `compc`. You must do this when the component skin consists of more than one ActionScript file, or you want to reskin more than one component.

The following example creates `myTheme.swc` from the `MyButtonSkin.as` and `MyCheckBoxIcon.as` files:

```
compc -flexlib $FLEX_HOME/apps/dev.war/WEB-INF/flex -root $FLEX_HOME/apps/
dev.war/skintest/ProgrammaticSkins -o myTheme.swc MyButtonSkin.as
MyCheckBoxIcon.as
```

For more information on using `compc`, see [“Using SWC files” on page 800](#).

Deploying programmatic skins

After you compile the theme SWC file, copy it to a directory that is accessible to the Flex application, but not in the Flex application's root.

Note: Do not store the theme file in the `flex_app_root/WEB-INF/flex/user_classes` directory or any directory specified by the `<lib-path>` child tag in the `flex-config.xml` file. Putting the theme's SWC file in one of these directories can cause versioning errors.

To use the new theme file in your application, point to it with the `theme` attribute of the `<mx:Application>` tag, as the following example shows:

```
<mx:Application theme="themes/myTheme.swc">
```

You can only apply a single theme file to each Flex application.

Debugging programmatic skins

You can develop and debug programmatic skins directly in Flex. To do this, you link them into your application without creating a SWC file. A SWC file is a compiled Flash movie that represents the skin. Debugging and editing it dynamically requires the extra steps of compiling and refreshing the SWC file.

To avoid compiling and deploying a SWC file after every change to the new skin, you can import skin ActionScript classes into your Flex applications. This makes quick edits, debugging, and testing easier, but requires that you maintain the files and `import` statements. So, you should only use this technique during development and testing and not during production. In production, you should compile the skins into a SWC file and deploy them as described in [“Deploying programmatic skins” on page 448](#).

To reskin Flex components without compiling a theme SWC file, store the skin source files in a directory accessible by the Flex application (but not the same directory as the application) and add an explicit `import` statement to link the skin classes into the application.

The following example imports two skin files, `MyButtonSkin.as` and `MyCheckBoxIcon.as`, which are in the `mytheme` subdirectory:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:Script>
    import mytheme.MyButtonSkin;
    import mytheme.MyCheckBoxIcon;
  </mx:Script>
```

You can edit and debug the skin files (in this example, `MyCheckBoxIcon.as` and `MyButtonSkin.as`) just like any local custom component. Changes to these files are reflected in the application when you reload it.

Although this appears to be a simpler method of deploying skins, it creates explicit dependencies that can break when you change the application. Thus, you should only use this technique when you are debugging applications and not during production.

After you finish debugging a skin, you can comment out the import and variable declarations in the MXML file, compile the skin's ActionScript file into a SWC file, and deploy the skin using the instructions in [“Deploying programmatic skins” on page 448](#).

Graphical skinning

Some skins cannot be edited in ActionScript. You must use Flash MX 2004 to edit these purely graphical skins. By using Flash MX 2004, you can use all tools that are available to the Flash environment to create complex and appealing skins.

When you edit graphical skins in Flash MX 2004, it is best to modify the existing symbols rather than delete and recreate them, because all dependencies have been established in the samples. If you delete and recreate a symbol, you must ensure that its export identifier is correct and that it is included in the master skin symbol.

When working with the sample graphical skin files in Flash MX 2004, you should not edit any ActionScript attached to the symbols. This code creates the proper linkage between the components and graphical skins.

Navigating graphical skin samples

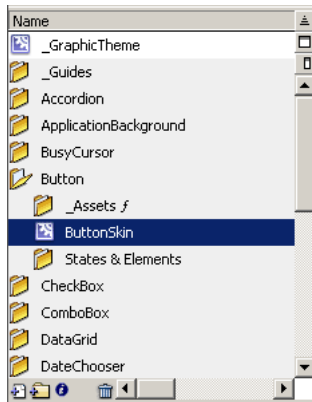
Flex provides several sample graphical skin files that you can use to graphically reskin Flex components. These sample themes are called Pulse. Flex includes the `pulseBlue fla` and `pulseOrange fla` files that define themes that you can use as a basis for reskinning components. These files are located in the `flex_install_dir/resources/themes/graphics` directory.

If you open one of the Pulse FLA files, you will notice that each skinnable graphical component has a folder in the Library with the same name as the component. Within that folder in the Library is the master skin symbol and all the graphical symbols that make up the skin. For example, the Button control's folder contains the ButtonSkin master skin symbol at the top level of the folder.

Each folder also contains the following subdirectories:

- **States & Elements** Contains categorized graphical symbols for each of the component's states. For example, the Button skin contains a number of symbols for the Down, Up, Over, and Disabled states.
- **_Assets f** Contains the raw image files and MovieClip objects that make up the component skins.

The following image shows the ButtonSkin master skin symbol inside the Button skin symbol folder:



When you export a skin as a theme SWC file, you export the master skin symbol, not other symbols in the library. For a Button control, you export the ButtonSkin symbol as a SWC file. You can put multiple master skin symbols in a single SWC file; see [“Exporting multiple symbols as a SWC file” on page 452](#).

Reskinning graphical skins

To reskin a component with graphical skins, you edit the graphics within the Flash MX 2004 environment and export the symbol as a SWC file. You then use the SWC file as a theme in your Flex applications. Deploying the SWC file for a graphical skin is the same as deploying one for a programmatic skin. The difference is how you generate the SWC file.

When working in the Flash MX 2004 environment, you should understand how the Library is used to organize symbols and graphical assets. In addition, you should understand how layers are used to build various graphics from multiple pieces. For more information about working in the Flash environment, see the Flash MX 2004 documentation.

You can use the following general process for editing component skins in Flash MX 2004.

To reskin a component in Flash MX 2004:

1. Open the Pulse sample FLA file in Flash MX 2004.
2. Find the symbol in the Library that you want to reskin.
3. Select the appropriate symbol in the Library and edit the graphic on the stage. Ensure that you do not edit any associated ActionScript in the FLA file.
4. Right click the component's master skin symbol, and select Export SWC File. For example, if you edited the DataSortArrow symbol for the DataGrid component, export the DataGridAssets master skin symbol to a SWC file.

The Export File dialog box appears.

5. Name the new SWC file anything you want.
6. After you compile the SWC file, copy it to a directory that is accessible to the Flex application, but not the application's root directory.

Note: Do not store the theme file in the *flex_app_root*/WEB-INF/flex/user_classes directory or any directory specified by the `<lib-path>` child tag in the flex-config.xml file. Putting the theme's SWC file in one of these directories can cause versioning errors.

7. Reference the file with the theme attribute of the `<mx:Application>` tag, as the following example shows:

```
<mx:Application theme="themes/myTheme.swc">
```

Creating a new theme file

Flex provides two example FLA files that you can use as a basis for your reskinned component theme files. However, sometimes you might want to start a new theme file. In this example, you create an empty FLA file. To add components to the FLA file, you copy them from one of the sample FLA files, edit the graphics, and export the new skins as a theme SWC file. You use this SWC file in your Flex applications.

To create a new theme file:

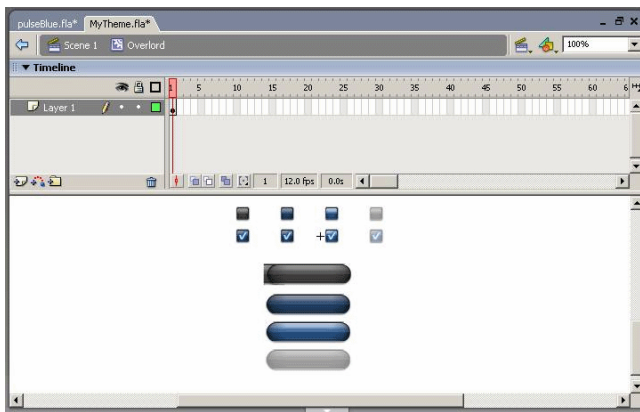
1. Start the Flash MX 2004 environment.
2. Create a FLA file by selecting File > New. Then select Flash Document.
3. Open the Library of the new theme file by selecting Window > Library. This library should be empty.
4. Open one of the sample Pulse FLA files; for example, pulseBlue fla. These files are located in the *flex_install_dir*/resources/themes/graphic directory.
5. Open the Library of the sample Pulse theme file by selecting Window > Library.
6. Drag the folder for the component that you want to reskin from the Pulse Library to the new file's Library.
7. Edit the skin symbols in your new file.
8. Save the new FLA file and export the component's master skin symbol as a SWC file.

Exporting multiple symbols as a SWC file

Sometimes you will want to create a new theme SWC file that contains master skin symbols for more than a single symbol. For example, you might want to graphically reskin both the Button and CheckBox components. Flex only lets you reference a single theme file in the `<mx:Application>` tag, so you must create a single SWC file from multiple master skin symbols. This section describes how to do this.

To export multiple symbols as a SWC file:

1. Create a new FLA file.
2. Copy the necessary components from the Pulse Library to the new FLA file's Library.
3. Drag the components that you want to reskin from the new FLA file's Library to the stage. For example, if you dragged the CheckBox and Button symbols onto the stage, your stage might look like the following:



4. Select all the symbols on the stage that you want to add to your new SWC file.
5. Right click while you have all the graphics selected, and select Convert to Symbol.
The Convert To Symbol dialog box appears.
6. Give the new symbol a name and select MovieClip for Behavior. Flash creates a new symbol in the Library.
You will export this new symbol that contains references to all the other symbols in order to create a new SWC file.
7. Enter Symbol Editing mode for the new symbol.
8. Edit the graphics that make up the component skins as you would any other graphical skin files.
9. Export the new symbol as a SWC file.

You can use the new SWC file as a theme file in the same way that you would use any other SWC file. It contains references to all the master skin symbols that you added.

CHAPTER 18

Using Behaviors

Behaviors let you add animation and motion to your application in response to some user or programmatic action. For example, you can use behaviors to cause a dialog box to bounce slightly when it receives focus, or to slowly fade in when it becomes visible.

This chapter describes how to build behaviors into your applications and also describes the two parts of a behavior: triggers and effects.

Contents

| | |
|---|-----|
| Applying behaviors. | 453 |
| Customizing an effect | 464 |
| Defining a custom effect | 468 |
| Defining and playing an effect in ActionScript. | 471 |
| Using a custom effect trigger | 472 |

Applying behaviors

A behavior is a combination of a trigger paired with an effect. A *trigger* is an action, such as a mouse click on a component, a component getting focus, or a component becoming visible. An *effect* is a visible or audible change to the component that occurs over a period of time, measured in milliseconds. Examples of effects are fading, resizing, or moving a component. You can define multiple effects for a single trigger.

For example, a pet store application might contain a button for each pet category. When the user clicks a button, a window that contains breed names becomes visible. As the window becomes visible, it moves to the bottom left corner of the screen, and it grows from 100 x 100 pixels to 300 x 300 pixels.

By default, Macromedia Flex components do not play an effect when a trigger occurs. To configure a component to use an effect, you associate an effect with the trigger. For example, you could define an effect that causes a button to slowly fade in when it becomes visible.

Note: Triggers are not events. For example, a Button control has both a `mouseDownEffect` trigger and a `mouseDown` event. The trigger initiates an effect when a user performs a mouse click on a component. You use the `mouseDown` event to specify the event handler that is executed when the user clicks the component.

About triggers and effects

Flex uses a Cascading Style Sheet (CSS) to define a trigger that you reference as a property of an MXML tag, within the `<mx:Style>` tag, or in an ActionScript function. The CSS property of a trigger uses the following naming convention:

triggerEffect

where *trigger* is the trigger name. For example, the `focusIn` trigger occurs when a component gains focus; the CSS property name for the `focusIn` trigger is `focusInEffect`. The `focusOut` trigger occurs when a component loses focus; its CSS property name is `focusOutEffect`.

The following table lists the CSS style name that corresponds to each trigger:

| CSS style name | Triggering event |
|-------------------------------------|---|
| <code>creationCompleteEffect</code> | Component is created. |
| <code>focusInEffect</code> | Component gains keyboard focus. |
| <code>focusOutEffect</code> | Component loses keyboard focus. |
| <code>hideEffect</code> | Component becomes invisible by changing the <code>visible</code> property of the component from <code>true</code> to <code>false</code> . |
| <code>mouseDownEffect</code> | User presses the mouse button while the mouse pointer is over the component. |
| <code>mouseOutEffect</code> | User rolls the mouse pointer off of the component. |
| <code>mouseOverEffect</code> | User rolls the mouse pointer over the component. |
| <code>mouseUpEffect</code> | User releases the mouse button. |
| <code>moveEffect</code> | Component is moved. |
| <code>resizeEffect</code> | Component is resized. |
| <code>showEffect</code> | Component becomes visible by changing the <code>visible</code> property of the component from <code>false</code> to <code>true</code> . |

Applying an effect in MXML

You use the CSS property name as a property of an MXML tag to configure a behavior. For example, to configure the effect for the `mouseDown` trigger in an `<mx:Button>` tag, you use the following MXML syntax:

```
<mx:Button id="myButton" mouseDownEffect="WipeLeft" />
```

A `WipeLeft` effect makes the Button control appear as if it were being wiped onto the Macromedia Flash Player stage from right to left. Flex also supports the `WipeRight`, `WipeUp`, and `WipeDown` effects. For more information, see [“List of effects” on page 460](#).

You can also apply an effect in an `<mx:Style>` tag. For example, to configure the effect for the hide trigger in all of the `<mx:Button>` tags in an application, you can use the following MXML syntax:

```
<mx:Style>
  Button{
    mouseDownEffect: WipeLeft;
  }
</mx:Style>
```

When you set an effect trigger in a component tag, that setting overrides settings you make in an `<mx:Style>` tag.

Applying an effect in ActionScript

Because Flex implements effect triggers as styles, you can use the `setStyle()` and `getStyle()` methods to apply effects.

The code in the following example alternates the Fade and WipeLeft effects for the `mouseDownEffect` style of a Button control:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">

  <mx:Script>
    <![CDATA[
      function changeEffect(){
        if (myButton.getStyle("mouseDownEffect") == "WipeLeft") {
          myButton.setStyle("mouseDownEffect","Fade");
        }
        else if (myButton.getStyle("mouseDownEffect") == "Fade") {
          myButton.setStyle("mouseDownEffect","WipeLeft");
        }
      }
    ]]>
  </mx:Script>

  <mx:Effect>
    <mx:Fade name="Fade" duration="1000"/>
  </mx:Effect>

  <mx:Button id="myButton" click="changeEffect();" label="My Button"
    mouseDownEffect="WipeLeft" mouseUpEffect="WipeRight" />
</mx:Application>
```

Using the effectStart and effectEnd events

Every user interface component has an `effectStart` and an `effectEnd` event. Event handlers you assign to these events execute when an effect starts or ends, respectively.

The event object passed to the handler for these effects contains the following properties:

- `target` A reference to the component that uses the effect.
- `type` The string `effectStart` or `effectEnd`.
- `effect` A reference to the effect.

For an example, see [“Zooming a component above 100 percent” on page 458](#).

Disabling container layout for effects

By default, Flex updates the layout of a container’s children when a new child is added to it, when a child is removed from it, when a child is resized, and when a child is moved. Because the Move effect modifies a child’s position, and the Zoom effect modifies a child’s size and position, they both cause the container to update its layout.

However, when the container updates its layout, it can actually reverse the results of the effect. For example, you use the Move effect to reposition a container child. At some time later, you change the size of another container child, forcing the container to update its layout. This layout update can cause the child that moved to be sent back to its original position.

To prevent Flex from performing layout updates, you can set the `autoLayout` property of a container to `false`. Its default value is `true`, which configures Flex to always update layouts. You always set the `autoLayout` property on the parent container of the component using the effect. For example, if you want to control the layout of a child of a Grid container, you set the `autoLayout` property for the parent `GridItem` container of the child, not for the Grid container.

You set the `autoLayout` property to `false` when you use a Move effect in parallel with a Resize or Zoom effect. You must do this because the Resize or Zoom effect can cause an update to the container’s layout, which can return the child back to its original location.

When you use the Zoom effect on its own, you might decide to set the `autoLayout` property to `false`, or you may leave it with its default value of `true`. For example, if you use a Zoom effect with the `autoLayout` property set to `true`, as the child grows or shrinks, Flex automatically updates the layout of the container to reposition its children based on the new size of the child. If you use a Zoom effect with the `autoLayout` property set to `false`, the child resizes around its center point and the remaining children do not change position.

The HBox container in the following example uses the default vertical alignment of `top` and the default horizontal alignment of `left`. If you apply a Zoom effect to the image, the HBox container resizes to hold the image, and the image remains aligned with the upper left corner of the container:

```
<mx:HBox>
  <mx:Image source="myImage.jpg" />
</mx:HBox>
```

In the next example, the image is centered in the HBox container. If you apply a Zoom effect to the image, as it resizes, it remains centered in the HBox container.

```
<mx:HBox horizontalAlign="center" verticalAlign="middle">
  <mx:Image source="myImage.jpg" />
</mx:HBox>
```


By default, the size of the HBox container is big enough to hold the image at its original size. If you disable layout updates, and use the Zoom effect to enlarge the image, or use the Move effect to reposition the image, the image might extend past the boundaries of the HBox container, as the following example shows:

```
<mx:HBox autoLayout="false">
  <mx:Image source="myImage.jpg" />
</mx:HBox>
```

Since you set the `autoLayout` property to `false`, the HBox container does not resize as the image resizes. If the image grows to a size so that it extends beyond the boundaries of the HBox container, the container adds scroll bars and clips the image at its boundaries.

To prevent the scroll bars from appearing, you can use the `height` and `width` properties to explicitly size the HBox container so that it is large enough to hold the modified image, or set the `clipContent` property of the container to `false` so that the image can extend past its boundaries.

Improving performance when resizing Panel containers

When you apply a Resize effect to a Panel container, the measurement and layout algorithm for the effect executes on every frame of the animation. When a Panel container has many children, the animation can be jerky because Macromedia Flex cannot update the screen quickly enough. Also, resizing one Panel container often causes other Panel containers to resize as well.

To solve this problem, you can use the Resize effect's `hideChildren` property to hide the children of Panel containers while the Resize effect is playing. The value of the `hideChildren` property is an Array of Panel containers that should include the Panel containers that resize during the animation. Before the Resize effect plays, Flex iterates through the Array and hides the children of each of the specified Panel containers.

Because the MXML syntax for effects does not currently support Array properties, you cannot use the `hideChildren` property with an effect that is declared using the `<mx:Resize>` tag. You must trigger the effect in ActionScript.

In the following example, the children of the `panelOne` and `panelTwo` Panel containers are hidden while the Panel containers resize:

```
<mx:HBox>
  <mx:VBox>
    <mx:Panel id="panelOne" mouseDown="runEffect()">...</mx:Panel>
    <mx:Panel id="panelTwo">...</mx:Panel>
  </mx:VBox>
  <mx:Panel id="panelThree">...</mx:Panel>
</mx:HBox>

<mx:Script>
  function runEffect() {
    var e = new mx.effects.Resize(panelOne);
    e.heightFrom = 100;
    e.heightTo = 200;
    e.hideChildren = [panelOne, panelTwo];
    e.playEffect();
  }
</mx:Script>
```

Note: The Flex Store application included in the samples.war file contains a more complex example that uses the Resize effect's `hideChildren` property. You can extract the samples.war file to your application server.

For each Panel container in the `hideChildren` Array, the following effect triggers execute:

- `resizeStartEffect` Delivered before the Resize effect begins playing.
- `resizeEndEffect` Delivered after the Resize effect finishes playing.

If the `resizeStartEffect` trigger specifies an effect to play, the Resize effect is delayed until the effect finishes playing.

The default value for the Panel container's `resizeStartEffect` and `resizeEndEffect` triggers is `Dissolve`, which plays the Dissolve effect. For more information about the Dissolve effect, see [“List of effects” on page 460](#).

To disable the Dissolve effect so that a Panel container's children are hidden immediately, you must set the value of the `resizeStartEffect` and `resizeEndEffect` triggers to `none`.

Zooming a component above 100 percent

By default, when you use the Zoom effect with a `zoomTo` value that is greater than 100, it causes undesirable layering of components. You can work around this by calling the zoomed component's `swapDepths()` method to manually change the depth of the component when zooming it and restore its original size. When you swap the depth of the component, you can enlarge it over other components in the application.

The `popToTop()` function in the following example changes the depth of a zoomed component when the `zoomTo` value is greater than 100; the `popToTop()` function is specified as the `effectStart` event handler for a Button control. The `restore()` method resets a zoomed component to its original depth; the `restore()` method is specified as the `effectEnd` event handler for the Button control.

This example also sets the `autoLayout` property to `false` to prohibit the Tile container from updating the layout during zooming, and sets the `clipContent` property to `false` to let the Button controls extend beyond the boundaries of the Tile container.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:Script>
    <![CDATA[
      var dp:Array = [1, 2, 3, 4, 5, 6, 7, 8, 9];

      function popToTop(target:mx.core.UIObject){
        var parent = target.parent;
        var topSibling = parent.getChildAt(parent.numChildren-1);

        if (topSibling != target)
          target.setDepthAbove(topSibling);
      }

      function restore(target:mx.core.UIObject){
        var parent = target.parent;
        var childIndex = parent.getChildIndex(target);
```

```

        if (childIndex < parent.numChildren - 1)
            target.setDepthBelow(parent.getChildAt(childIndex+1));
    }
]]>
</mx:Script>

<mx:Effect>
    <mx:Zoom name="myZoom" zoomFrom="100" zoomTo="175" />
    <mx:Zoom name="cut" zoomTo="100" duration="1" />
</mx:Effect>

<mx:Tile backgroundColor="#A9C365" marginLeft="3" marginRight="3"
    marginTop="3" marginBottom="3" horizontalGap="3" verticalGap="3"
    autoLayout="false" clipContent="false" >

    <mx:Repeater dataProvider="{dp}">
        <mx:Button height="49" width="50" label="hi!"
            mouseOver="popToTop(event.target)"
            mouseOverEffect="myZoom"
            mouseOut="restore(event.target)"
            mouseOutEffect="cut"/>
    </mx:Repeater>
</mx:Tile>
</mx:Application>

```

List of effects

The following table lists the effects that Flex supports. This table includes the MXML tag definition for each effect. You use the tag to customize effect properties. For more information, see [“Customizing an effect” on page 464](#).

| Effect | MXML tag | Description |
|----------|--|--|
| Dissolve | <pre><mx:Dissolve name="ID" alphaFrom="val" alphaTo="val" color="val" duration="val" easing="funcName" suspendBackground Processing="val" /></pre> | <p>Animate the component from transparent to opaque, or from opaque to transparent.</p> <p>The Dissolve effect has the following properties:</p> <ul style="list-style-type: none">• <code>name</code> Specifies the effect identifier.• <code>alphaFrom</code> Specifies the initial alpha level (0=completely hidden, 100=fully opaque). If omitted, Flex uses the component's current alpha level.• <code>alphaTo</code> Specifies the final alpha level.• <code>color</code> Specifies hex value of the color of the rectangle that floats over the target object while the effect is playing.• <code>duration</code> Specifies the effect duration. The default value is 500 ms.• <code>easing</code> Specifies an easing function, which lets you change the speed of an animation; for more information, see “Easing functions” on page 466.• <code>suspendBackgroundProcessing</code> If true, blocks all background processing, such as measurement and layout and responses from data services, while the effect is playing. <p>If you specify the <code>Dissolve</code> effect for the show or hide trigger, and if you omit the <code>alphaFrom</code> and <code>alphaTo</code> properties, the effect automatically transitions from 0 to 100 for a show trigger, and 100 to 0 for a hide trigger.</p> <p>The effect does the following when it is played:</p> <ul style="list-style-type: none">• When the effect begins, an opaque rectangle is created. The rectangle floats above the target object. Its color matches the value of the <code>Dissolve.color</code> property, and its <code>alpha</code> value is initially set to $(100 - \text{Dissolve.alphaFrom})$.• As the effect plays, the alpha of the rectangle animates from $(100 - \text{alphaFrom})$ to $(100 - \text{alphaTo})$. As the rectangle becomes more and more opaque, the content underneath it gradually disappears.• When the effect finishes, the rectangle is destroyed. <p>If the target object is a container, only the children of the container dissolve. The borders do not dissolve.</p> |

| Effect | MXML tag | Description |
|--------|--|---|
| Fade | <pre><mx:Fade name="ID" alphaFrom="val" alphaTo="val" duration="val" easing="funcName" suspendBackground Processing="val" /></pre> | <p>Animate the component from transparent to opaque, or from opaque to transparent.</p> <p>The Fade effect has the following properties:</p> <ul style="list-style-type: none"> • <code>name</code> Specifies the effect identifier. • <code>alphaFrom</code> Specifies the initial alpha level (0=transparent, 100=fully opaque). If omitted, Flex uses the component's current alpha level. • <code>alphaTo</code> Specifies the final alpha level. • <code>duration</code> Specifies the effect duration. The default value is 500 ms. • <code>easing</code> Specifies an easing function, which lets you change the speed of an animation. For more information, see “Easing functions” on page 466. • <code>suspendBackgroundProcessing</code> If <code>true</code>, blocks all background processing, such as measurement and layout and responses from data services, while the effect is playing. <p>If you specify the <code>Fade</code> effect for the show or hide trigger, and if you omit the <code>alphaFrom</code> and <code>alphaTo</code> properties, the effect automatically transitions from 0 to 100 for a show trigger, and 100 to 0 for a hide trigger.</p> <p>Note: To use the Fade effect with text, you must use an embedded font, not a device font. For more information, see Chapter 16, “Using Styles and Fonts,” on page 395.</p> |

| Effect | MXML tag | Description |
|--------|--|--|
| Move | <pre><mx:Move name="ID" xFrom="val" yFrom="val" xTo="val" yTo="val" xBy="val" yBy="val" duration="val" easing="funcName" suspendBackground Processing="val" /></pre> | <p>Changes the position of a component over a specified time interval.</p> <p>The Move effect has the following properties:</p> <ul style="list-style-type: none"> <code>name</code> Specifies the effect identifier. <code>xFrom</code> and <code>yFrom</code> Specify the initial position of the component. If omitted, Flex uses the current position. <code>xTo</code> and <code>yTo</code> Specify the destination position. <code>xBy</code> and <code>yBy</code> Specify the number of pixels to move the component in the x and y directions. Values can be positive or negative. <code>duration</code> Specifies the effect duration. The default value is 500 ms. <code>easing</code> Specifies an easing function, which lets you change the speed of an animation; for more information, see “Easing functions” on page 466. <code>suspendBackgroundProcessing</code> If <code>true</code>, blocks all background processing, such as measurement and layout and responses from data services, while the effect is playing. <p>For the <code>xFrom</code>, <code>xTo</code>, and <code>xBy</code> properties, you can specify any two of the three values; Flex calculates the third. If you specify all three properties, Flex ignores the <code>xBy</code> property. The same is true for the <code>yFrom</code>, <code>yTo</code>, and <code>yBy</code> properties.</p> <p>If you specify a Move effect for a move trigger, and if you do not set the six From, To, and By properties, Flex sets them to create a smooth transition between the object’s old position and its new position.</p> <p>When the Move effect runs, the layout around the component that is moving does not readjust. Setting a container’s <code>autoLayout</code> property to <code>true</code> has no effect on this behavior.</p> |
| Pause | <pre><mx:Pause name="ID" duration="val" suspendBackground Processing="val" /></pre> | <p>Does nothing for a specified period of time. This effect is useful when you need to composite effects. For more information, see “Composite effects” on page 465.</p> <p>The Pause effect has the following properties:</p> <ul style="list-style-type: none"> <code>name</code> Specifies the effect identifier. <code>duration</code> Specifies the effect duration. The default value is 500 ms. <code>suspendBackgroundProcessing</code> If <code>true</code>, blocks all background processing, such as measurement and layout and responses from data services, while the effect is playing. |

| Effect | MXML tag | Description |
|--------|--|--|
| Resize | <pre><mx:Resize name="ID" widthFrom="val" heightFrom="val" widthTo="val" heightTo="val" widthBy="val" heightBy="val" duration="val" easing="funcName" suspendBackgroundProcessing="val" /></pre> | <p>Changes the width and height of a component over a specified time interval.</p> <p>The Resize effect has the following properties:</p> <ul style="list-style-type: none"> <code>name</code> Specifies the effect identifier. <code>widthFrom</code> and <code>heightFrom</code> Specify the initial width and height. If omitted, Flex uses the current size. <code>widthTo</code> and <code>heightTo</code> Specify the final width and height. <code>widthBy</code> and <code>heightBy</code> Specify the number of pixels to modify the size as either a positive or negative number relative to the initial width and height. <code>duration</code> Specifies the effect duration. The default value is 500 ms. <code>easing</code> Specifies an easing function, which lets you change the speed of an animation; for more information, see “Easing functions” on page 466. <code>suspendBackgroundProcessing</code> If true, blocks all background processing, such as measurement and layout and responses from data services, while the effect is playing. <p>For <code>widthFrom</code>, <code>widthTo</code>, and <code>widthBy</code>, you can specify any two of the three values, and Flex calculates the third. If you specify all three, Flex ignores <code>widthBy</code>. The same is true for <code>heightFrom</code>, <code>heightTo</code>, and <code>heightBy</code>.</p> <p>If you specify a Resize effect for a resize trigger, and if you do not set the six From, To, and By properties, Flex sets them to create a smooth transition between the object’s old size and its new size.</p> <p>When you apply a Resize effect, the layout manager resizes neighboring components based on the size changes to the target component. To run the effect without resizing other components, place the target component in a Canvas container.</p> <p>When you use the Resize effect with Panel containers, you can hide Panel children to improve performance. For more information, see “Improving performance when resizing Panel containers” on page 457.</p> |

| Effect | MXML tag | Description |
|---|---|--|
| WipeLeft WipeRight WipeUp WipeDown | <pre><mx:WipeXXX name="ID" show="boolean" duration="val" easing="funcName" suspendBackground Processing="val" /></pre> | <p>Defines a bar wipe effect. The before or after state of the component must be invisible.</p> <p>These effects have the following properties:</p> <ul style="list-style-type: none"> • name Specifies the effect identifier. • show If <code>true</code> (default), causes the component to appear. If <code>false</code>, causes the component to disappear. • duration Specifies the effect duration. The default value is 500 ms. • easing Specifies an easing function, which lets you change the speed of an animation; for more information, see “Easing functions” on page 466. • suspendBackgroundProcessing If <code>true</code>, blocks all background processing, such as measurement and layout and responses from data services, while the effect is playing. <p>If you specify a Wipe effect for a show or hide trigger, by default, Flex sets the <code>show</code> property to <code>true</code> if the component is invisible, and <code>false</code> if the component is visible.</p> |
| Zoom | <pre><mx:Zoom name="ID" zoomFrom="val" zoomTo="val" duration="val" easing="funcName" suspendBackground Processing ="val" /></pre> | <p>Zooms a component in or out from its center point.</p> <p>The Zoom effect has the following properties:</p> <ul style="list-style-type: none"> • name Specifies the effect identifier. • zoomFrom Specifies a number that represents the scale at which to start the zoom. The default value is 0. • zoomTo Specifies a number to zoom to. The default value is 100. The maximum value is also 100. • duration Specifies the effect duration. The default value is 500 ms. • easing Specifies an easing function, which lets you change the speed of an animation; for more information, see “Easing functions” on page 466. • suspendBackgroundProcessing If <code>true</code>, blocks all background processing, such as measurement and layout and responses from data services, while the effect is playing. <p>To use a <code>zoomTo</code> value greater than 100, you must swap the depth of the component being zoomed so that it appears above other components in the application. For more information, see “Customizing an effect” on page 464.</p> |

Customizing an effect

Every effect accepts at least one property that you can use to configure it. In addition, all effects take the following properties:

- **duration** Specifies the time, in milliseconds, over which the effect occurs.
- **easing** (except Pause) Specifies a pointer to an easing function, which lets you change the speed of an animation; for more information, see [“Easing functions” on page 466](#).

For a complete list of effect properties, see [“List of effects” on page 460](#).

To customize an effect:

1. Add an `<mx:Effect>` tag near the top of the MXML document.
2. Inside the `<mx:Effect>` tag, insert the MXML tag for the effect that you want to modify.
3. Set the name property of the MXML tag for the effect to a unique name.

When using the modified effect in an MXML application, refer to it by its name property.

4. Set the properties of the new effect tag.

For more information on tag properties, see [“List of effects” on page 460](#).

Custom effects

The following example creates two new versions of the Fade effect. The `SlowFade` effect uses a two-second duration; the `ReallySlowFade` effect uses an eight-second duration:

```
<!-- Customize the Fade effect to create two new effects. -->
<mx:Effect>
  <mx:Fade name="SlowFade" duration="2000"/>
  <mx:Fade name="ReallySlowFade" duration="8000"/>
</mx:Effect>

<!-- Use the built-in Fade effect for this label.-->
<mx:Image creationCompleteEffect="Fade" source="first.jpeg" />

<!-- Use custom versions of the Fade effect for these labels. -->
<mx:Image creationCompleteEffect="SlowFade" source="second.jpeg" />
<mx:Image creationCompleteEffect="ReallySlowFade" source="third.jpeg" />
```

The following example creates three versions of the Move effect:

```
<mx:Effect>
  <mx:Move name="SmallMove" xBy="5" yBy="5" duration="1000"/>
  <mx:Move name="MediumMove" xBy="20" yBy="20" duration="1000"/>
  <mx:Move name="LargeMove" xBy="50" yBy="50" duration="1000"/>
</mx:Effect>
```

Composite effects

Flex supports two methods to combine, or *composite*, effects: parallel and sequence. When you combine multiple effects in parallel, the effects play at the same time. When you combine multiple effects in sequence, one effect must complete before the next effect starts.

To define parallel or sequential effects, include the `<mx:Parallel>` or `<mx:Sequence>` tag in the `<mx:Effect>` tag. The `<mx:Parallel>` and `<mx:Sequence>` tags accept only a name property.

The following example defines the parallel effect `WipeRightUp`, which combines the `WipeRight` and `WipeUp` effects:

```
<mx:Effect>
  <mx:Parallel name="WipeRightUp">
    <mx:WipeRight duration="1000" />
    <mx:WipeUp duration="1000" />
  </mx:Parallel>
</mx:Effect>
```

```

<mx:VBox id="myBox" hideEffect="WipeRightUp" >
  <mx:TextArea id="aTextArea" text="hello" />
</mx:VBox>

<mx:Button id='myButton' click="myBox.visible = !myBox.visible;"
  label="Wipe!" />

```

The click handler for the Button control alternates turning the VBox container visible and invisible. When the VBox container becomes invisible, it uses the WipeRightUp effect as its hide effect.

The following example shows a sequence effect that moves and resizes a component:

```

<mx:Effect>
  <mx:Sequence name="MoveResize">
    <mx:Move xBy="20" yBy="20" duration="1000"/>
    <mx:Resize heightBy="20" widthBy="20" duration="1000"/>
  </mx:Sequence>
</mx:Effect>

```

You can nest `<mx:Parallel>` and `<mx:Sequence>` tags inside each other. For example, two effects can run in parallel, followed by a third effect running in sequence.

Easing functions

You can change the speed of an animation by defining an easing function for an effect. With easing, you can create a more realistic rate of acceleration and deceleration. You can also use an easing function to create a bounce effect or control other types of motion.

Note: An easing function takes four arguments, following the function signature popularized by Robert Penner. For more information, see www.ericd.net/chapter7.pdf.

The following code shows the format of an easing function:

```

function myEasingFunction(t, b, c, d) {
  ...
}

```

You specify the following arguments to an easing function:

- `t` specifies time.
- `b` specifies the initial position of a component.
- `c` specifies the total change in position of the component.
- `d` specifies the duration of the effect, in milliseconds.

In the following example, an easing function creates a bounce effect when combined with the Flex Move effect:

```

<mx:Script>
  <![CDATA[
    function myEasingFunction(t, b, c, d) {
      if ((t /= d) < (1 / 2.75)) {
        return c * (7.5625 * t * t) + b;
      }
      else if (t < (2 / 2.75)) {

```

```

        return c * (7.5625 * (t -= (1.5 / 2.75)) * t + .75) + b;
    }
    else if (t < (2.5 / 2.75)) {
        return c * (7.5625 * (t -= (2.25 / 2.75)) * t + .9375) + b;
    }
    else {
        return c * (7.5625 * (t -= (2.625 / 2.75)) * t + .984375) + b;
    }
    };
    ]]>
</mx:Script>

```

To use this easing function, you use the `<mx:Effect>` tag to define a custom effect, as the following example shows:

```

<mx:Effect>
    <mx:Move name="moveLeftShow" xFrom="600" xTo="0" yTo="0" duration="3000"
        easing="myEasingFunction" />
    <mx:Move name="moveRightHide" xFrom="0" xTo="600" duration="3000"
        easing="myEasingFunction" />
</mx:Effect>

```

In this example, you create two custom Move effects. The first effect moves a component to the left; the second effect moves it to the right. You can use these custom effects in a Flex application, as the following example shows:

```

<mx:LinkBar dataProvider="myVS" />
<mx:ViewStack id="myVS" borderStyle="solid">
    <mx:Canvas id="Canvas0" label="Canvas0"
        creationCompleteEffect="moveLeftShow"
        showEffect="moveLeftShow"
        hideEffect="moveRightHide" >
        <mx:Box height="300" width="600" backgroundColor="#00FF00">
            <mx:Label text="Screen 0" color="#FFFFFF" fontSize="40"/>
        </mx:Box>
    </mx:Canvas>
    <mx:Canvas id="Canvas1" label="Canvas1"
        showEffect="moveLeftShow" hideEffect="moveRightHide" >
        <mx:Box height="300" width="600" backgroundColor="#0033CC">
            <mx:Label text="Screen 1" color="#FFFFFF" fontSize="40"/>
        </mx:Box>
    </mx:Canvas>
</mx:ViewStack>

```

In this example, you use the custom effects in the `showEffect` and `hideEffect` properties of the children of a `ViewStack` container. When you click a label in the `LinkBar` navigator container, the corresponding child of the `ViewStack` container slides in from the right, and bounces to a stop against the left margin of the `ViewStack` container, while the previously visible child of the `ViewStack` container slides off to the right.

The custom effect for the `showEffect` property is only triggered when the child's visibility changes from `false` to `true`. Therefore, the first child of the `ViewStack` container also includes a `creationCompleteEffect` property. This is necessary to trigger the effect when Flex first creates the component. If you omit the `creationCompleteEffect` property, you will not see the `moveLeftShow` effect when the application starts.

Defining a custom effect

To define a custom effect, you create a subclass of the `mx.effects.Effect` class. You can also create a subclass of one of the standard effects included with Flex; these effects are all subclasses of the `mx.effects.Effect` class. The class should contain a `playEffect()` method to start the effect, and can optionally contain an `endEffect()` method to stop the effect.

The following example shows an effect class that uses a `Sound` object to play an embedded MP3 file. This class directly extends the `mx.effects.Effect` class.

```
// MySound.as
class MySound extends mx.effects.Effect
{
    [Embed(mimeType="audio/mpeg",source="sample.mp3")]
    var soundSymbol:String;

    var s:Sound;

    public function playEffect():Void{
        super.playEffect();
        s = new Sound();
        s.attachSound(soundSymbol);
        s.start();
    }

    public function endEffect():Void{
        s.stop();
    }
}
```

To declare a custom effect class in an MXML file, you place an effect tag with the same name as the effect class inside an `<mx:Effect>` tag. You reference the custom effect the same way you reference a standard effect.

The following example shows an application that uses the `MySound` effect. The `MySound.as` class file and the `sample.mp3` file are in the same directory as the MXML file.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    xmlns:local="*">

    <!-- Declare the SoundEffect effect. -->
    <mx:Effect>
        <local:MySound name="mySoundEffect"/>
    </mx:Effect>

    <!-- Use the SoundEffect effect with a mouseOver trigger. -->
    <mx:Label fontSize="20" text="Chime right in!"
        mouseOverEffect="mySoundEffect" />
</mx:Application>
```

The following example shows a custom visual effect that rotates a component. The effect extends the `mx.effects.TweenEffect` class.

```
// Rotate.as
import mx.effects.Tween;
```

```

/* Rotate extends TweenEffectAnimate the rotation of the component */
[Style(name="angleFrom",type="Number")]
[Style(name="angleTo",type="Number")]

/** Rotate a component from its center point.
<p><b>MXML Syntax</b></p>
<p>You use the &lt;Rotate> tag to define a Rotate effect. The &lt;Rotate> tag accepts the properties in the following syntax example.</p>
<p>
<PRE>
&lt;Rotate<br>
    rotateFrom="30"<br>
    rotateTo="60"<br>
    styleName="<i>Style name; no default.</i>"<br>
    duration="500"<br>
    easing="<i>Easing function name; no default.</i>"<br>
/>
</PRE>
</p>
Let (phi) be angle between r=(Ox,Oy - Cx,Cy) and -X Axis.
(theta) be clockwise further angle of rotation.
Xtheta = Cx - rCos(theta + phi);
Ytheta = Cy - rSin(theta + phi);
Xtheta = Cx - rCos(theta)Cos(phi) + rSin(theta)Sin(phi);
Ytheta = Cy - rSin(theta)Cos(phi) - rCos(theta)Sin(phi);

Now Cos(phi) = w/2r; Sin(phi) = h/2r;

Xtheta = Cx - rCos(theta)Cos(phi) + rSin(theta)Sin(phi);
Ytheta = Cy - rSin(theta)Cos(phi) - rCos(theta)Sin(phi);
Xtheta = Cx - rCos(theta)w/2r + rSin(theta)h/2r;
Ytheta = Cy - rSin(theta)w/2r - rCos(theta)h/2r;
Xtheta = Cx - wCos(theta)/2 + hSin(theta)/2;
Ytheta = Cy - wSin(theta)/2 - hCos(theta)/2;
*/

class Rotate extends mx.effects.TweenEffect
{
    var className:String = "Rotate";
    // absolute points of rotation
    var centerX:Number;
    var centerY:Number;
    // relative points of rotation
    [Inspectable]
    var aboutX:Number;
    [Inspectable]
    var aboutY:Number;

    [Inspectable(defaultValue=0)]
    var angleFrom:Number;
    [Inspectable(defaultValue=360)]
    var angleTo:Number;

    function Rotate(targetObj:Object){
        target = targetObj;

```

```

    }

    function playEffect():Void{
        super.playEffect();
        var radVal:Number = Math.PI * target._rotation/180;
        if(aboutX == undefined){
            aboutX = target.width/2;
        }

        if(aboutY == undefined){
            aboutY = target.height/2;
        }

        // Find the about point
        centerX = target._x + (aboutX)*Math.cos(radVal) -
        (aboutY)*Math.sin(radVal);
        centerY = target._y + (aboutX)*Math.sin(radVal) +
        (aboutY)*Math.cos(radVal);

        if(angleFrom == undefined){
            angleFrom = target._rotation;
        }

        if(angleTo == undefined){
            angleTo = (target._rotation == 0) ? ((angleFrom > 180) ? 360 : 0) :
            target._rotation ;
        }

        tween = new Tween(this, angleFrom, angleTo, duration);

        if (easing)
            tween.setEasingEquation(target, easing);

        // Set back to initial position before the screen refreshes
        target._rotation = angleFrom;
        radVal = Math.PI * angleFrom/180;

        target._x = centerX - (aboutX)*Math.cos(radVal) +
        (aboutY)*Math.sin(radVal);
        target._y = centerY - (aboutX)*Math.sin(radVal) -
        (aboutY)*Math.cos(radVal);
    }

    function onTweenUpdate(val):Void{
        target._rotation = val;
        var radVal:Number = Math.PI * val/180;

        target._x = centerX - (aboutX)*Math.cos(radVal) +
        (aboutY)*Math.sin(radVal);
        target._y = centerY - (aboutX)*Math.sin(radVal) -
        (aboutY)*Math.cos(radVal);
    }

    function getAffectedProperties():Array{
        return ["_rotation"];
    }

```

```

    }
}

```

The following example shows an application that uses the Rotate effect:

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">

    <!-- Declare the Rotate effect.-->
    <mx:Effect>
        <local:Rotate name="myRotateEffect" angleFrom="0" angleTo="100"
            duration="5000" xmlns:local="*" />
    </mx:Effect>

    <mx:Button id="panel" label="Rotate" mouseOverEffect="myRotateEffect" />
</mx:Application>

```

Defining and playing an effect in ActionScript

You can declare and play the effect in an event handler function. This is very useful for using a control to trigger an effect on another control. For example, the event handler in the following example is applied to a Button control's click event to trigger a Resize effect on an Image control:

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" width="800"
    height="518">
    <mx:Script>
        <![CDATA[
            function myEffectHandler(){
                // Create a Resize effect and apply it to a TextArea control named
                // myText.
                var resizeLarge = new mx.effects.Resize(myText);
                // Set resized width and height, and effect duration.
                resizeLarge.widthTo=150;
                resizeLarge.heightTo=150;
                resizeLarge.duration=750;
                // Play the effect.
                resizeLarge.playEffect();
            }
        ]]>
    </mx:Script>

    <mx:Canvas height="300" width="500" borderStyle="solid">
        <mx:Button x="50" y="50" click="myEffectHandler();" />
        <mx:TextArea x="100" y="100" id="myText" text="Here is some text." />
    </mx:Canvas>
</mx:Application>

```

You can also create an event handler that combines effects into a composite effect and then plays the composite effect, as the following example shows:

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" width="800"
    height="518">
    <mx:Script>

```

```

<![CDATA[
    function myEffectHandler(){
        // Create a Resize effect.
        var resizeLarge = new mx.effects.Resize(myText);

        // Set resized width and height, and effect duration.
        resizeLarge.widthTo=150;
        resizeLarge.heightTo=150 ;
        resizeLarge.duration=750;

        var moveToRight = new mx.effects.Move(myLabel);
        moveToRight.xTo=200;
        moveToRight.duration=750;

        var resizeAndMove = new mx.effects.Parallel();
        resizeAndMove.addChild(resizeLarge);
        resizeAndMove.addChild(moveToRight);

        // Play the composite effect.
        resizeAndMove.playEffect();
    }
]]>
</mx:Script>

<mx:Canvas height="300" width="500" borderStyle="solid">
    <mx:Button x="50" y="50" click="myEffectHandler();"/>
    <mx:TextArea x="100" y="100" id="myText" text="Here is some text."/>
</mx:Canvas>
</mx:Application>

```

Using a custom effect trigger

You can create a custom effect trigger to handle situations for which standard triggers do not meet your needs. For example, suppose you want to apply an effect that sets the brightness level of a component. The standard `showEffect` and `hideEffect` properties are paired with a component's the `visible` property. As a result, you can only trigger the effect specified in a `hideEffect` property by setting a component's `visible` property to `false`. When a component's `visible` property is set to `false`, the component should be invisible, so `showEffect` and `hideEffect` are not appropriate triggers for setting a component to different levels of brightness.

The following example shows a custom `Button` control that dispatches two events, `darken` and `brighten`, based on changes to the `bright` property. The control also defines two effect triggers, `darkenEffect` and `brightenEffect`; when you declare an event `[Event("eventname")]`, you can create a corresponding effect by declaring `[Effect("eventnameEffect")]`.

```

<?xml version="1.0"?>

<!-- MyButton.mxml -->
<mx:Button xmlns:mx="http://www.macromedia.com/2003/mxml" width="200"
    height="200">

    <mx:Metadata>
        [Event("darken")]
    
```



```

    [Event("brighten")]
    [Effect("darkenEffect")]
    [Effect("brightenEffect")]
</mx:Metadata>

<mx:Script>
    var _bright:Boolean = true;

    function set bright(val:Boolean) {
        _bright = val;

        if (val)
            dispatchEvent({type:"brighten"});
        else
            dispatchEvent({type:"darken"});
    }

    function get bright():Boolean {
        return _bright;
    }
</mx:Script>

</mx:Button>

```

The application file in the following example contains a `MyButton` control. The `darkenEffect` and `brightenEffect` properties are set to the `FadeOut` and `FadeIn` effects, respectively. The `click` property of another `Button` control toggles the `MyButton` control's `bright` property and executes the corresponding effect (`FadeOut` or `FadeIn`).

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" xmlns:local="*"
    height="800" width="800">

    <mx:Effect>
        <mx:Fade name="FadeOut" alphaFrom="100" alphaTo="20"/>
        <mx:Fade name="FadeIn" alphaFrom="20" alphaTo="100"/>
    </mx:Effect>

    <local:MyButton id="btn" darkenEffect="FadeOut" brightenEffect="FadeIn"/>

    <mx:Button click="btn.bright = !btn.bright"/>
</mx:Application>

```


CHAPTER 19

Creating Charts in Flex

The ability to display data in a chart or graph can make data interpretation much easier for Macromedia Flex application users. Rather than present a simple table of numeric data, you can display a bar, pie, line, or other type of chart using colors, captions, and a two-dimensional representation of your data.

This chapter describes the Flex charting controls and describes how you use them to create charts.

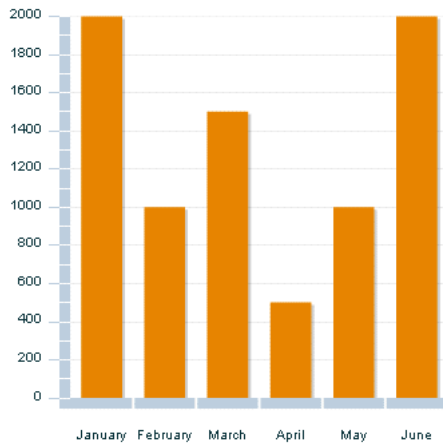
Contents

| | |
|-------------------------------------|-----|
| About charting. | 475 |
| Defining chart data | 483 |
| Creating charts. | 487 |
| Formatting chart elements | 500 |
| Showing DataTips | 509 |
| Using images in charts | 512 |
| Using fills. | 514 |
| Using Legend controls | 521 |
| Stacking charts. | 526 |
| Handling user interactions. | 527 |
| Using effects with charts | 531 |
| Creating custom charts | 536 |

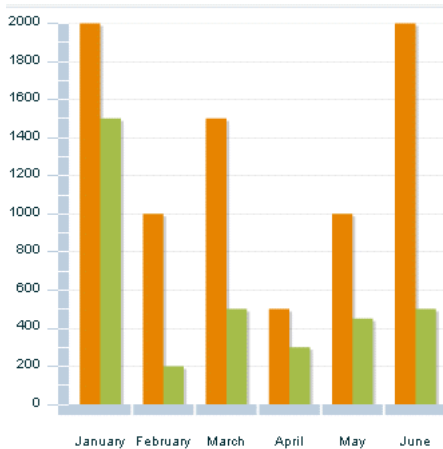
About charting

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 provides you with a great deal of control over the appearance of the 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 the percentage growth in sales over four business quarters:



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 growth and one for profit growth:



Using the charting controls

Flex includes a set of charting controls that lets you create some of the most common chart types, and also gives you a large amount of control over the appearance of your charts. The charting controls are located in the `mx.charts.*` package. The following table lists the supported chart types:

| Chart type | Chart control | Chart series |
|------------|---------------|--------------|
| Area | AreaChart | AreaSeries |
| Bar | BarChart | BarSeries |
| Bubble | BubbleChart | BubbleSeries |
| Column | ColumnChart | ColumnSeries |
| Line | LineChart | LineSeries |
| Pie | PieChart | PieSeries |
| Plot | PlotChart | PlotSeries |

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 text for the chart axes, and other properties specific to the chart type.

The chart series determines the data from the data provider that the chart displays. You can use the chart series to specify a single data series, or multiple series. You can also use the chart series to define the appearance of each data series in the 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"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:Panel title="My Chart" >
    <mx:AreaChart id="chart" dataProvider="{dataSet}" >
      <mx:series>
        <mx:Array>
          <mx:AreaSeries yField="Sales" >
            <mx:stroke>
              <mx:Stroke color="0x9A9A00" weight="2" />
            </mx:stroke>
            <mx:fill>
              <mx:SolidColor color="0x7EAEFF"/>
            </mx:fill>
          </mx:AreaSeries>
        </mx:Array>
      </mx:series>
    </mx:AreaChart>
  </mx:Panel>
</mx:Application>
```

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

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

```
<mx:AreaChart id="chart" dataProvider="{dataSet}" >
  <mx:series>
    <mx:Array>
      <!-- First data series. -->
      <mx:AreaSeries yField="Sales" >
        <mx:stroke>
          <mx:Stroke color="0x9A9A00" weight="2" />
        </mx:stroke>
        <mx:fill>
          <mx:SolidColor color="0x7EAEFF"/>
        </mx:fill>
      </mx:AreaSeries>
      <!-- Second data series. -->
      <mx:AreaSeries yField="Revenue" >
        <mx:stroke>
          <mx:Stroke color="0x9A9A00" weight="2" />
        </mx:stroke>
        <mx:fill>
          <mx:SolidColor color="0xAA0000" alpha="30" />
        </mx:fill>
      </mx:AreaSeries>
    </mx:Array>
  </mx:series>
</mx:AreaChart>
```

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

```
<mx:BarChart id="bc0" dataProvider="{expenses}" width="100%" height="100%">
```

Parent containers of the charting control must also be set using percent values, otherwise the chart does not resize when the window resizes.

Using a renderer

Chart controls use *renderers* to draw the chart elements. The chart elements are the part of the chart that appears within the x-axis and y-axis of the chart. For a bar chart, the renderer controls the shape, color, and fill of the bars. A pie chart does not have axes, so the renderer draws the entire chart.

Each chart control has a default renderer, and all data series in that chart use the default renderer. You can also specify a different renderer for each data series in the chart. To change a renderer, you add a `<mx:renderer>` child tag to the series tag. Renderers also affect the shape, color, and fill of any Legend markers that are generated from that series.

The chart created by the following code example uses the default `CircleRenderer` for the first data series. The `CircleRenderer` is the default renderer for the data series in a `PlotChart` control. The second series uses a `DiamondRenderer`. As a result, the plotted points of the second series use diamond shapes to indicate each data point.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
```

```

<mx:Panel title="My Plot" >
  <mx:PlotChart id="chart" dataProvider="{dataSet}" >
    <mx:series>
      <mx:Array>
        <!-- First series uses default renderer. -->
        <mx:PlotSeries xField="Newton" yField="NewYork" radius="16" />
        <!-- Second series uses DiamondRenderer. -->
        <mx:PlotSeries yField="SanFrancisco" xField="Miami" radius="7">
          <mx:renderer>
            <mx:DiamondRenderer />
          </mx:renderer>
        </mx:PlotSeries>
      </mx:Array>
    </mx:series>
  </mx:PlotChart>
</mx:Panel>
</mx:Application>

```

You can use any renderer with a chart control as long as that renderer is compatible with the chart. For example, the column chart requires that its renderer implements the `BoxRenderer` interface. The `SimpleBoxRenderer` class implements that interface, as do the `CircleRenderer`, `CrossRenderer`, `DiamondRenderer`, `TriangleRenderer`, and `ShadowBoxRenderer` classes. Therefore, you can use any of these renderers with a column chart.

The following table shows the chart type, series, default renderer, and the interface that a renderer must implement to be used with a chart:

| Chart | Chart series | Default renderer | Renderer interface |
|-------------|--------------|---------------------|--|
| AreaChart | AreaSeries | SimpleAreaRenderer | AreaRenderer |
| BarChart | BarSeries | SimpleBoxRenderer | BoxRenderer |
| BubbleChart | BubbleSeries | CircleRenderer | BoxRenderer |
| ColumnChart | ColumnSeries | SimpleBoxRenderer | BoxRenderer |
| LineChart | LineSeries | SimpleLineRenderer | LineRenderer |
| PieChart | PieSeries | ShadowWedgeRenderer | WedgeRenderer |
| PlotChart | PlotSeries | CircleRenderer | BoxRenderer (rotates renderers if you have more than one series) |

In some cases, you can use multiple types of data series in a single chart. For example, if you create a `CartesianChart` control, you can use both an `AreaSeries` and a `BarSeries`. In this case, you can use all the renderers supported by those series in the same chart. For more information on `CartesianChart` controls, see [“Creating custom charts” on page 536](#).

Flex provides additional renderers that are not used as default renderers. You can use these to change the appearance of your charts by adding shadows or graphics. The following table lists the available renderers for each chart type and their interface:

| Chart type | Interface | Available renderers |
|-------------|------------------|--|
| AreaChart | AreaRenderer | SimpleAreaRenderer |
| BarChart | BoxRenderer | AssetRenderer |
| BubbleChart | | CircleRenderer |
| ColumnChart | | CrossRenderer |
| PlotChart | | DiamondRenderer |
| | | ShadowBoxRenderer |
| | | SimpleBoxRenderer |
| | TriangleRenderer | |
| LineChart | LineRenderer | SimpleLineRenderer ShadowLineRenderer |
| PieChart | WedgeRenderer | ShadowWedgeRenderer |

The behavior of most renderers is self-explanatory. The SimpleBoxRenderer draws elements in the shape of boxes. The DiamondRenderer draws elements in the shape of diamonds. The ShadowBoxRenderer and ShadowLineRenderer add shadows to the elements that they draw. The AssetRenderer, however, inserts an image file or SWF file that takes the place of the element's shape. For more information, see [“Using images in charts” on page 512](#).

About the axes

Chart *data* is defined by the fields specified in the chart's series objects. The appearance and contents of *axis labels* are defined by the `<mx:horizontalAxis>` (x-axis) and `<mx:verticalAxis>` (y-axis) tags and those tags' renderers (`<mx:horizontalAxisRenderer>` and `<mx:verticalAxisRenderer>`). These tags define not only the data ranges that appear in the chart, but also map the data points to their names and labels. This latter mapping has a large impact on how Flex renders the values of DataTips, axis labels, and tick marks.

You can define an axis as being either a `LinearAxis` or a `CategoryAxis`. A `LinearAxis` maps numeric data to the axis. You use the `<mx:LinearAxis>` child tag of the `<mx:horizontalAxis>` or `<mx:verticalAxis>` tag to customize the range of values displayed along the axis, and to set the increment between the axis labels of the tick marks.

A `CategoryAxis` maps a set of discrete 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 are not necessarily numeric.

Depending on the chart type, you usually specify a `CategoryAxis` for the horizontal or vertical axis, and a `LinearAxis` for the other axis. For example, a column chart usually uses a `CategoryAxis` along the horizontal axis to label the columns. The vertical axis defines the height of the columns and so requires a `LinearAxis` definition. With a bar chart, the horizontal axis defines the length of the columns and, therefore, should be defined as the `LinearAxis`.

In many cases, you are only required to define one axis as being a `LinearAxis` or a `CategoryAxis`. Flex assumes that all axes are of type `LinearAxis`. However, to use decorations such as `DataTip` labels and legends, you might be required to explicitly define both axes.

For a plot chart, both axes are considered a `LinearAxis`, since the data point is the intersection of two coordinates. As a result, you are not required to specify either axis, although you can still do so to provide additional settings, such as minimum and maximum values. As with all charts, in `PlotChart` controls, you use the `name` property of the data series to define legend labels.

Each axis has a corresponding `AxisRenderer` (`horizontalAxisRenderer` and `verticalAxisRenderer`) that defines the appearance of axis labels, tick marks, and the titles of those axes. In addition to defining formats, you can use an `AxisRenderer` to customize the value of the axis labels. For more information, see “[Formatting chart elements](#)” on page 500.

By default, Flex calculates the labels that appear along the x-axis and y-axis of the chart, based on the chart type and orientation. For example, a column chart has the following default values:

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

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

For a bar chart, which is a column chart rotated 90 degrees, reverse the descriptions for the x-axis and y-axis to determine the default labels.

The following sections describe the `LinearAxis` and `CategoryAxis` in more detail.

About the `LinearAxis`

A `LinearAxis` maps numeric data (such as sales volume, revenue, or profit) to coordinates on the screen. This axis defines the height or length of the bar or line in charts like the `BarChart`, `ColumnChart`, `LineChart`, and `AreaChart` controls.

A `LinearAxis` lets you set additional information, such as minimum and maximum values, that appear along the axis. These values change how the data is rendered on the screen.

For a `LinearAxis`, you can set the interval at which tick marks appear along the axis and determine whether the axis starts at zero. You do this by using the `baseAtZero` property. Set this property to `true` to make the axis start at zero; set it to `false` to let Flex determine a reasonable starting value relative to the values along the axis.

About the `CategoryAxis`

A `CategoryAxis` maps discrete categorical data (for example, 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` attribute of the `<mx:CategoryAxis>` tag 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. However, a `CategoryAxis` used in a chart does not inherit its `dataProvider` property from the containing chart, so you must explicitly set the `dataProvider` property on a `CategoryAxis`.

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

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

If the data provider contains an array of labels only, you do not need to specify the `categoryField` attribute.

You can customize the labels of the `<mx:CategoryAxis>` tag rather than use the axis labels in the data provider. To do this, set the value of the `CategoryAxis`'s `dataProvider` property to a custom array of labels, as the following example shows:

```
<mx:Script>
  var myCategories : Array = ["a", "b", "c", "d", "e", "f", "g"];
</mx:Script>
<mx:AreaChart>
  <mx:horizontalAxis>
    <mx:CategoryAxis dataProvider="{myCategories}" />
  </mx:horizontalAxis>
  ...
</mx:AreaChart>
```

For more information about chart data providers, see [“Defining chart data” on page 483](#).

Setting ranges of tick marks

By default, Flex sets tick marks alongside the axes. Flex determines the minimum and maximum tick-mark values and sets the interval based on the settings of the `LinearAxis`. You can override the values that Flex calculates. By changing the range of tick marks, you also change the range of the data displayed in the chart.

To set ranges of tick marks, you use the `<mx:LinearAxis>` child tag of the `<mx:verticalAxis>` or `<mx:horizontalAxis>` tag. The following table describes the attributes that affect tick marks:

| Attribute | Description |
|-----------------------|--|
| <code>minimum</code> | The lowest value of the axis. |
| <code>maximum</code> | The highest value of the axis. |
| <code>interval</code> | The number of units between tick marks along the axis. |

The following example defines the range of the y-axis:

```
<mx:AreaChart id="chart">
  <mx:verticalAxis>
    <mx:LinearAxis minimum="0" maximum="300" interval="15" />
  </mx:verticalAxis>
  ...
</mx:AreaChart>
```

In this example, the minimum value displayed along the y-axis is 0, the maximum value is 300, and the interval is 15. Therefore, the label text is 0, 15, 30, 45, and so on.

For information about setting the length and location of tick marks, see [“Formatting tick marks” on page 507](#).

About charting events

The chart controls include new events to accommodate user interaction with data points in charts. These events are described in [“Handling user interactions” on page 527](#).

Defining chart data

The chart controls all take a `dataProvider` property that defines the data for the chart. A *data provider* is a collection of objects, similar to an array. The chart components use a flat, or list-based, data provider, similar to a one-dimensional array.

A data provider consists of two parts: a collection of data objects and an API. The data provider API is a set of methods and properties that a class must implement so that a Flex component recognizes it as a data provider.

The data provider creates a level of abstraction between Flex components and the data that you use to populate them. You can populate multiple components from the same data provider, switch data providers for a component at runtime, and modify the data provider so that changes are reflected by all components using the data provider.

You can also bind properties to a data provider. All other properties of chart controls do not let you bind to them unless they are inherited from the `UIObject` or `UIComponent` or another class outside of the `mx.charts.*` package.

For more information about using data providers in Flex, see [“About data providers” on page 91](#).

Using chart data

To use the data from a data provider in your chart control, you point 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, if 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, 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 horizontal axis's `categoryField` property:

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

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

When using chart data, keep the following in mind:

- You must match a series with a data provider field if you want to display that series.
- Some series use only one field from the data provider, while others can use two or more. For example, you only specify `field` for a `PieSeries`, while you can specify an `xField` and a `yField` for a `PlotSeries` and an `xField`, `yField`, and `radiusField` for a `BubbleSeries`.
- Most of the series can determine suitable defaults for their nonprimary dimensions if no field is specified. For example, `Column`, `Line`, and `Area` series map the data to the chart's categories in the order it appears in the `dataProvider` if you do not explicitly set an `xField`. Similarly, a `BarSeries` maps the data to the categories if you do not set a `yField`.

For a complete list of which fields each data series can take, see the data series' entry in *Flex ActionScript and MXML API Reference*. For more information on data providers, see [“Using data provider controls” on page 33](#).

Generating chart data

To begin using the chart components, you need data that is available for charting, or you must create your own test data. To help you get started, the following example generates test data for use with the chart controls:

```
<mx:Script><![CDATA[
  // Define data provider array for the chart data.
  var dataSet:Array;
  // Define the number of elements in the array.
  var dsLength:Number = 10;
  function initApp() {
    // Initialize data provider array.
    dataSet = genData(dsLength);
  }
}]>
```

```

function genData(dpc) {
    var d:Array = [];
    // Create an object with four properties containing random values.
    var vals = {Newton:Math.random()*100, SanFrancisco:Math.random()*100,
        NewYork:Math.random()*100, Miami:Math.random()*100};
    // Create one data element for each label in categories.
    for (var i=0;i<dsLength;i++) {
        // Write the first data element to the data array.
        d.push(vals);
        var newVals = {};
        // Create a new data element with four properties.
        for(var aProp in vals) {
            newVals[aProp] = Math.max(0,vals[aProp] + Math.random()*10 - 5);
        }
        vals = newVals;
    }
    return d;
}
]]></mx:Script>

```

Each data provider array element contains four properties: Newton, SanFrancisco, NewYork, and Miami, and each property contains a random number.

You can use the data provider with the chart controls, as the following example shows:

```

<mx:PlotChart id="chart" dataProvider="{dataSet}" >
    <mx:series>
        <mx:Array>
            <mx:PlotSeries xField="Newton" yField="NewYork" radius="16" />
            <mx:PlotSeries yField="SanFrancisco" xField="Miami" radius="7" />
        </mx:Array>
    </mx:series>
</mx:PlotChart>

```

You can also define a model in MXML to populate a data provider, as the following example shows:

```

<mx:Model id="dataModel">
    <sampleData Newton="45" NewYork="23" SanFrancisco="47" Miami="80"/>
    <sampleData Newton="76" NewYork="81" SanFrancisco="62" Miami="101"/>
</mx:Model>
<mx:ColumnChart id="chart" dataProvider="{dataModel.sampleData}" />
    <mx:series>
        <mx:Array>
            <mx:ColumnSeries yField="Newton" />
        </mx:Array>
    </mx:series>
</mx:ColumnChart>

```

Changing chart data at runtime

You can change the fields of a series to change chart data at runtime. 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 chart control's series index, as the following example shows:

```
var mySeries:mx.charts.series.PieSeries = myChart.series[0];
series.yField = "Palimony";
```

The following example toggles the data series when the user clicks on the button:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<mx:Application initialize="initApp();"
  xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:Script><![CDATA[
    var dataSet : Array;
    var states = ["Wisconsin","Ohio","Arizona","Pennsylvania"];
    var curSeries:String;
    function initApp() {
      var newData = [];
      for(var i=0;i<states.length;i++) {
        newData[i]={ Apples: Math.floor(Math.random()*150),
                     Oranges: Math.floor(Math.random()*150),
                     state: states[i] }
      }
      dataSet = newData;
      curSeries = "apples";
    }
    function changeData() {
      var series:mx.charts.series.ColumnSeries = column.series[0];
      if (curSeries == "apples") {
        curSeries="oranges";
        series.yField = "Oranges";
      } else {
        curSeries="apples";
        series.yField = "Apples";
      }
    }
  ]]></mx:Script>
  <mx:ColumnChart id="column" dataProvider="{dataSet}" showDataTips="true" >
    <mx:horizontalAxis>
      <mx:CategoryAxis dataProvider="{dataSet}" categoryField="state" />
    </mx:horizontalAxis>
    <mx:series>
      <mx:Array>
        <mx:ColumnSeries yField="Apples" />
      </mx:Array>
    </mx:series>
  </mx:ColumnChart>
  <mx:Button id="b1" label="Change Series" click="changeData()" />
</mx:Application>
```

Creating charts

There are many different types of charts you can create from the same set of data. The following sections describe how to use the chart types in Flex:

- [“Area charts” on page 487](#)
- [“Bar charts” on page 489](#)
- [“Bubble charts” on page 490](#)
- [“Column charts” on page 491](#)
- [“Line charts” on page 492](#)
- [“Pie charts” on page 494](#)
- [“Plot charts” on page 498](#)

In addition to these chart types, you can also extend the CartesianChart control to create custom charts. For more information, see [“Creating custom charts” on page 536](#).

Area charts

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

The following figure shows an example of an area chart:



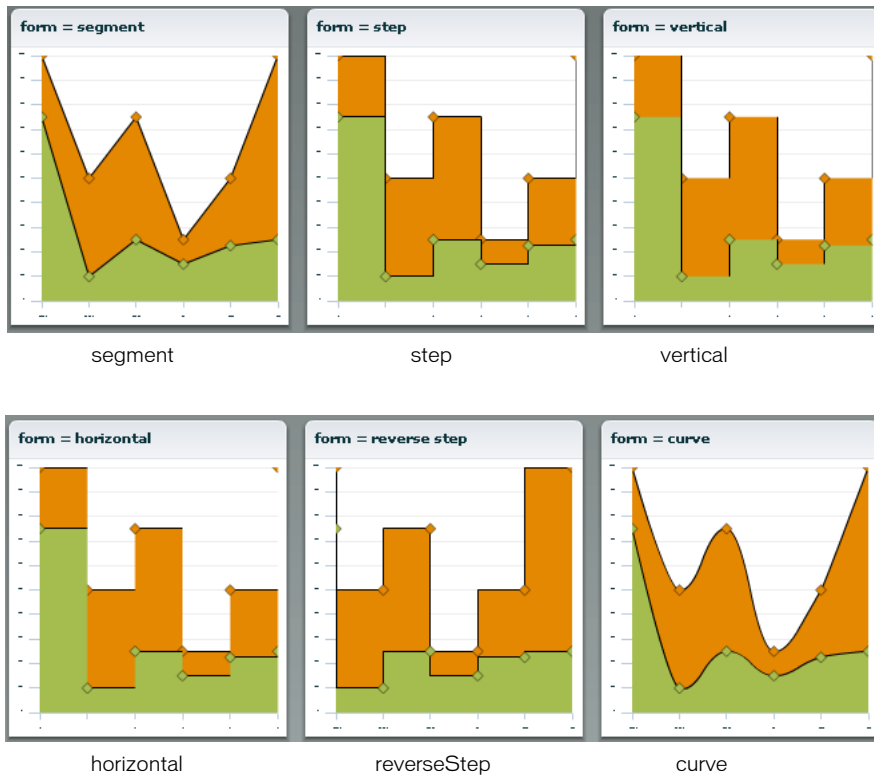
You can use the AreaChart control to represent a variety of chart variations, including overlaid, stacked, 100% stacked, and high-low areas.

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 [“Line charts” on page 492](#).

You use the `AreaSeries` with the `AreaChart` control to define the data for the chart. You commonly use the following properties of the `AreaSeries` to define your chart:

- `yField` Specifies the field of the data provider that determines the y-axis location of each data point.
- `xField` Specifies the field of the data provider that determines the x-axis location of each data point. If you omit this field, Flex arranges the data points in the order of the data in the data provider.
- `minField` Specifies the field of the data provider that determines the y-axis location of the bottom of an area. This field 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 minField” on page 512](#).
- `form` Specifies the way in which the data series are shown in the chart. Possible values are:
 - `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 only the vertical line from the y-coordinate of the first point to y-coordinate of the second point at the x-coordinate of the second point. Repeats this for each data point.
 - `horizontal` Draws only the horizontal line from the x-coordinate of the first point to x-coordinate of the second point at the y-coordinate of the first point. Repeats this for each data point.
 - `curve` Draws curves between data points.

The following image shows the available forms for an AreaSeries:



You can set the `type` property on an `AreaChart` control to stack and group series data in the chart. For more information, see [“Stacking charts” on page 526](#).

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 variety of chart variations, including clustered bars, overlaid stacked, 100% stacked, and high-low areas.

You use the `BarSeries` with the `BarChart` control to define the data for the chart. You commonly use the following properties of the `BarSeries` to define your chart:

- `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 minField” on page 512](#).

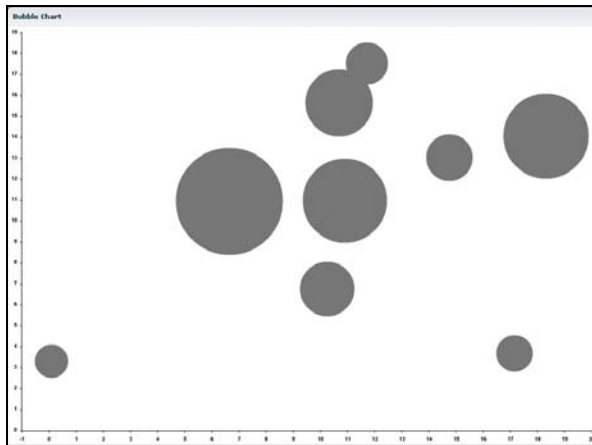
A bar chart is essentially a column chart rotated 90 degrees clockwise; therefore, bar charts and column charts share many of the same characteristics. For more information, see [“Column charts” on page 491](#).

Bubble charts

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

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

The following figure shows an example of a bubble chart:



You use the BubbleSeries with the BubbleChart control to define the data for the chart. You commonly use the following properties of the BubbleSeries to define your chart:

- `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"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:Script>
    var expenses = [{Month: "Jan", Profit: 2000, Expenses: 1500, 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 maxRadius="50" dataProvider="{expenses}"
        showDataTips="true">
        <mx:series>
            <mx:Array>
                <mx:BubbleSeries xField="Profit" yField="Expenses"
                    radiusField="Amount" />
            </mx:Array>
        </mx:series>
    </mx:BubbleChart>
</mx:Panel>
</mx:Application>

```

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 an example of a column chart, see [“About charting” on page 475](#).

You use the ColumnSeries with the ColumnChart control to define the data for the chart. You commonly use the following properties of the ColumnSeries to define your chart:

- **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 minField” on page 512](#).

The following example shows a ColumnChart control with two series:

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
    <mx:Script>
        var expenses = [{Month: "Jan", Profit: 2000, Expenses: 1500 },
                        {Month: "Feb", Profit: 1000, Expenses: 200},
                        {Month: "Mar", Profit: 1500, Expenses: 500}]];
    </mx:Script>
    <mx:Panel>
        <mx:ColumnChart id="column" dataProvider="{expenses}">
            <mx:horizontalAxis>
                <mx:CategoryAxis dataProvider="{expenses}" categoryField="Month" />
            </mx:horizontalAxis>
            <mx:series>
                <mx:Array>

```

```

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

```

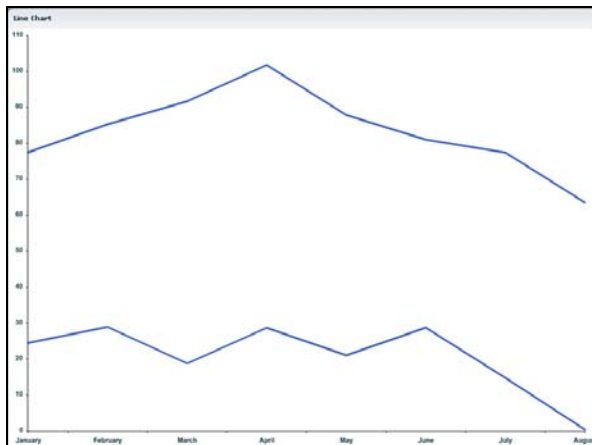
You can set the `type` property on a `ColumnChart` control to stack and group series data in the chart. For more information, see [“Stacking charts” on page 526](#).

You can customize the colors, sizes and positions of columns in a `ColumnChart` control. For more information, see [“Customizing bars and columns” on page 536](#).

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 figure shows an example of a simple line chart:

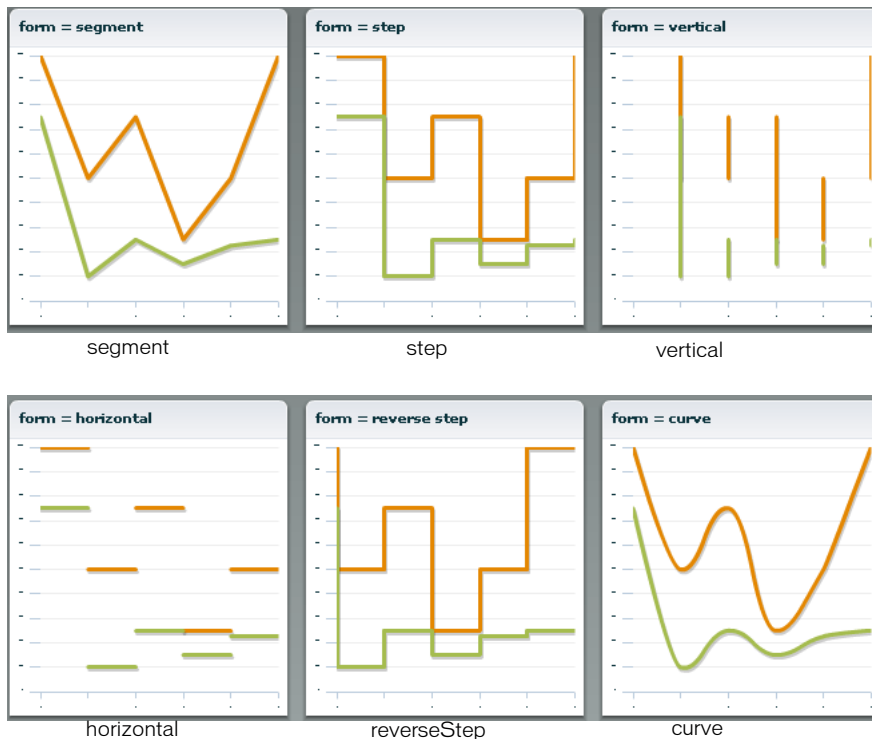


You use the `LineSeries` with the `LineChart` control to define the data for the chart. You commonly use the following properties of the `LineSeries` to define your chart:

- `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 are shown in the chart. Possible values are:
 - **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 only the vertical line from the y-coordinate of the first point to y-coordinate of the second point at the x-coordinate of the second point. Repeats this for each data point.
 - **horizontal** Draws only the horizontal line from the x-coordinate of the first point to x-coordinate of the second point at the y-coordinate of the first point. Repeats this for each data point.
 - **curve** Draws curves between data points.

The following image shows the available forms for a LineSeries:



The following example creates a LineChart control:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
```

```

<mx:Script>
    var expenses = [{Month: "Jan", Profit: 2000, Expenses: 1500, Amount: 450},
                    {Month: "Feb", Profit: 1000, Expenses: 200, Amount: 600},
                    {Month: "Mar", Profit: 1500, Expenses: 500, Amount: 300}];
</mx:Script>
<mx:Panel title="Line Chart">
    <mx:LineChart dataProvider="{expenses}" showDataTips="true">
        <mx:horizontalAxis>
            <mx:CategoryAxis dataProvider="{expenses}" categoryField="Month" />
        </mx:horizontalAxis>
        <mx:series>
            <mx:Array>
                <mx:LineSeries yField="Profit" name="Profit" />
                <mx:LineSeries yField="Expenses" name="Expenses" />
            </mx:Array>
        </mx:series>
    </mx:LineChart>
</mx:Panel>
</mx:Application>

```

You can remove the default shadows from the line by specifying a `SimpleLineRenderer` as the line renderer in the series definition. You can also specify the `ShadowLineRenderer` as the line series renderer, which draws a shadow under the line. The default is `ShadowLineRenderer`.

The following example creates a `LineChart` control whose line does not have a shadow:

```

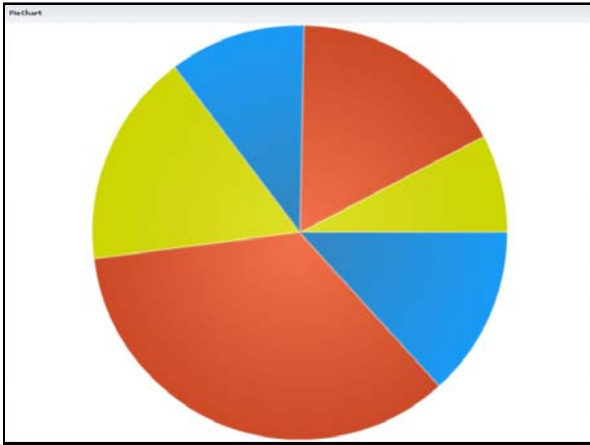
<mx:LineChart dataProvider="{expenses}" >
    <mx:horizontalAxis>
        <mx:CategoryAxis dataProvider="{expenses}" categoryField="Month" />
    </mx:horizontalAxis>
    <mx:series>
        <mx:Array>
            <mx:LineSeries yField="Amount">
                <mx:renderer>
                    <mx:SimpleLineRenderer />
                </mx:renderer>
            </mx:LineSeries>
        </mx:Array>
    </mx:series>
</mx:LineChart>

```

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 figure shows an example of a pie chart:



You use the `PieSeries` 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.

You commonly use the following properties of the `PieSeries` to define your chart:

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

The following example defines a `PieChart` control:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:Script>
    var expenses = [{Expense: "Taxes", Amount: 2000},
                    {Expense: "Rent", Amount: 1000},
                    {Expense: "Bills", Amount: 100},
                    {Expense: "Car", Amount: 450},
                    {Expense: "Gas", Amount: 100},
                    {Expense: "Food", Amount: 200}];
  </mx:Script>
  <mx:Panel>
    <mx:PieChart id="pie" dataProvider="{expenses}" showDataTips="true">
      <mx:series>
        <mx:Array>
          <mx:PieSeries field="Amount" nameField="Expense"
                        labelPosition="callout"/>
        </mx:Array>
      </mx:series>
    </mx:PieChart>
    <mx:Legend dataProvider="{pie}" />
  </mx:Panel>
</mx:Application>
```

Using labels

PieChart controls support labels that display information about each data point. All charts support DataTips, which display the value of a data point when the user moves the mouse over it. Labels are different from DataTips in that they are always visible and do not react to mouse movements. The PieChart control is the only chart that supports labels.

To add 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 is `none`.

The following table describes the valid values of the `labelPosition` property:

| Value | Description |
|--------------------------------|---|
| <code>callout</code> | 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. |
| <code>inside</code> | 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 <code>insideLabelSizeLimit</code> property, is hidden from view. |
| <code>insideWithCallout</code> | Draws labels inside the pie, but if labels are shrunk below a legible size, Flex converts them to callout labels. This is a common value to set <code>labelPosition</code> to when the actual size of your chart is flexible and users might resize it. |
| <code>none</code> | Does not draw labels. This is the default value. |
| <code>outside</code> | Draws labels around the boundary of the PieChart control. |

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

| Property | Description |
|-----------------------------------|--|
| <code>calloutGap</code> | Defines how much space, in pixels, to insert between the edge of the pie and the labels when rendering callouts. The default value is 10 pixels. |
| <code>calloutStroke</code> | Defines the line style used to draw the lines to call outs. For more information on defining line strokes, see “Setting stroke styles” on page 508 . |
| <code>insideLabelSizeLimit</code> | Defines the size threshold, expressed in points, below which inside labels are considered illegible. Below this threshold, labels are either removed entirely or turned into callouts based on the setting of the series’ <code>labelPosition</code> property. |

To modify the label text, you use the `labelFunction` property to specify a callback function. The function specified in `labelFunction` returns a string that Flex displays as the label over the pie wedge. The following is the required format of the callback function:

```
function function_name ( data, field, index, percentValue ) {}
```

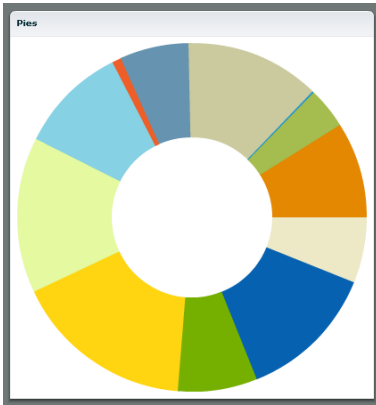
- `data` A reference to the data point that is represented by this pie wedge.
- `field` The field name from the data provider.
- `index` The number of the data point in the data provider.
- `percentValue` The size of the pie wedge relative to the pie. If the pie wedge is a quarter the size of the pie, this value is 25.

The following example generates labels that include data and formatting:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:Script><![CDATA[
    var expenses = [{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}];
    function display(data, field, index, percentValue) {
      return data.Expense + ": $" + data.Amount + newline + percentValue +
        "%";
    }
  ]]></mx:Script>
  <mx:Panel title="Expenditures for FY04">
    <mx:PieChart id="chart" dataProvider="{expenses}" showDataTips="false">
      <mx:series>
        <mx:Array>
          <mx:PieSeries labelPosition="callout" field="Amount"
            labelFunction="display" />
        </mx:Array>
      </mx:series>
    </mx:PieChart>
  </mx:Panel>
</mx:Application>
```

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:

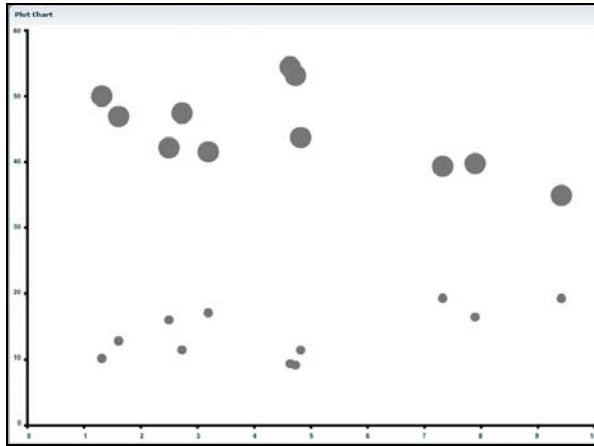
```
<mx:PieChart id="pie" dataProvider="{expenses}" innerRadius="30">
```

The value of the `innerRadius` property is a percentage value of the “hole” compared to the entire pie’s radius.

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 data series’ renderer.

The following figure shows an example of a plot chart with the CircleRenderer:



You use the PlotSeries class with the PlotChart control to define the data for the chart. You commonly use the following properties of the PlotSeries class to define your chart:

- **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. By default, the plot chart draws a circle at each data point with a radius of 5 pixels.

Note: Both the xField and yField properties are required for a PlotChart control.

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

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
<mx:Script>
    var expenses = [
        {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:PlotChart id="plot" dataProvider="{expenses}" showDataTips="true">
    <mx:series>
        <mx:Array>
            <mx:PlotSeries xField="Expenses" yField="Profit" name="Plot 1" />
            <mx:PlotSeries xField="Amount" yField="Expenses" name="Plot 2" />
            <mx:PlotSeries xField="Profit" yField="Amount" name="Plot 3" />
        </mx:Array>
    </mx:series>
</mx:PlotChart>
```

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

You can control the image that is displayed by the chart for each data point by setting the `renderer` of the series. 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 `renderer` for the chart (starting with diamond, then circle, then box).

The following example overrides the default `DiamondRenderer` for the series and uses circles instead:

```
<mx:PlotSeries xField="Expenses" yField="Profit" name="Plot 1" />
  <mx:renderer>
    <mx:CircleRenderer/>
  </mx:renderer>
</mx:PlotSeries>
```

Flex also selects a unique color for the fill and `renderer` of each series. You can override these with custom fills. For more information, see [“Using fills” on page 514](#).

Formatting chart elements

You can style the appearance of almost all chart elements, from the font properties of an axis label to the width of the stroke in a legend.

This section describes the following topics:

- [“Applying styles to chart elements” on page 500](#)
- [“Setting chart margins” on page 502](#)
- [“Adding axis titles” on page 504](#)
- [“Defining axis labels” on page 505](#)
- [“Rotating axis elements” on page 506](#)
- [“Formatting tick marks” on page 507](#)
- [“Formatting axis lines” on page 508](#)
- [“Setting stroke styles” on page 508](#)

Applying styles to chart elements

To set style properties in your chart controls, you can either use Cascading Style Sheet (CSS) syntax or set the style properties inline as tag attributes. This section describes these methods of formatting charts. 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 419](#).

Applying chart styles with CSS

You can use the control name in CSS definitions to define styles for that control. For example, the following style definition defines the font for the BubbleChart control:

```
<mx:Style>
  BubbleChart {
    font-family: labelFont;
    font-size: 8;
    color: 0xFFFF00;
  }
</mx:Style>
```

The chart controls are subclasses of the Flex UIObject and UIComponent classes. Therefore, these controls support the style properties defined by those classes, including the style properties for defining the font face, font size, and other font characteristics of the controls.

Some styles, such as font-size and font-family, 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 *Flex ActionScript and MXML API 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 using either a class selector on the axis renderer or predefined axis style properties.

Using predefined axis style properties

There are two predefined axis style properties: `horizontalAxisStyle` and `verticalAxisStyle`. Flex applies each style to the appropriate axis.

The following example removes tick marks from the horizontal axis:

```
<mx:Style>
  .myColumnChart {
    horizontalAxisStyle: myHorizAxisStyle;
  }
  .myHorizAxisStyle {
    tickPlacement: none;
  }
</mx:Style>
<mx:ColumnChart styleName="myColumnChart" dataProvider="{dataSet}" >
  ...
</mx:ColumnChart>
```

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` to point to that class selector.

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

```
<mx:Style>
  .MyStyle {
    font-size:7;
  }
</mx:Style>
<mx:ColumnChart dataProvider="{expenses}" showDataTips="true" >
  ...
  <mx:horizontalAxisRenderer>
    <mx:AxisRenderer title="Number of Apples" styleName="MyStyle" />
  </mx:horizontalAxisRenderer>
  ...
</mx:ColumnChart >
```

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 395](#).

Applying styles inline

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

The following example sets the `tickPlacement` property of the horizontal axis to `none`:

```
<mx:horizontalAxisRenderer>
  <mx:AxisRenderer tickPlacement="none" />
</mx:horizontalAxisRenderer>
```

You can also access the properties of renderers in `ActionScript` so that you can change the format during runtime. For additional information about axis renderers, see [“Using a renderer” on page 478](#).

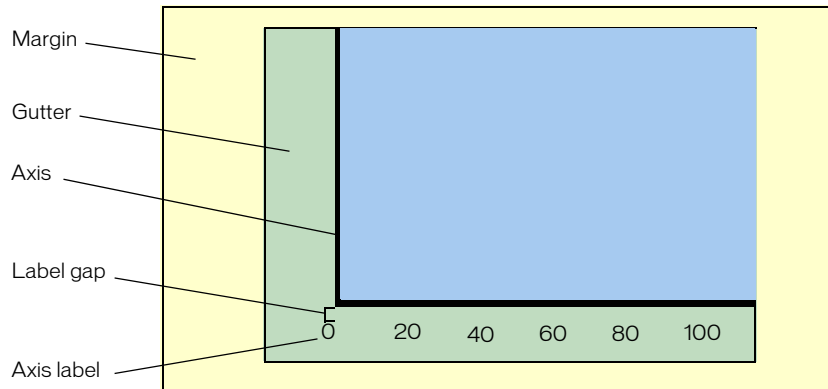
Setting chart margins

As with other Flex components, the margins of a chart define an area between the outside bounds of the control and its content. Flex draws nothing other than the background fill in the margins.

You can set the margin values for a chart control using the `marginLeft` and `marginRight` properties that it inherits from the `UIObject` class. You can also use the `marginTop` and `marginBottom` properties that chart controls inherit from the `ChartBase` class.

Flex charts also have gutters. The gutter is the area between the margin and the actual axis lines. Flex draws axis labels, titles, and tick marks in the gutter of the chart. Charts adjust the gutters to accommodate these axis decorations, but you can specify explicit gutter values.

The following graphic shows the locations of the gutters and margins on a chart:



The following styles 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 chart rendering, 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` styles. It also sets the `marginTop` property to 20.

```
<mx:Style>
    ColumnChart {
        gutterLeft: 50;
        gutterRight: 50;
        gutterBottom: 50;
        marginTop: 20
    }
</mx:Style>
<mx:ColumnChart ... />
```

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

```
<mx:ColumnChart dataProvider="{expenses}" gutterLeft="50" gutterRight="50"
    gutterBottom="50" gutterTop="20">
```

In addition to the gutter and margin 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`, as the following example shows:

```
<mx:ColumnChart dataProvider="{expenses}" showDataTips="true">
    <mx:horizontalAxisRenderer>
```

```

        <mx:AxisRenderer labelGap="20"/>
    </mx:horizontalAxisRenderer>
    ...
</mx:ColumnChart >

```

Adding axis titles

Axes can include a title that describes the purpose of the axis to the users. Flex does not add titles to the chart axes unless you explicitly set them. To add titles to the axes of a chart, you use the **title attribute of the `<mx:AxisRenderer>` child tag of the `<mx:verticalAxisRenderer>` or `<mx:horizontalAxisRenderer>` tag.** If the axis title is for a vertical axis, you must embed the font that is used for the axis.

The following example sets the title of the horizontal axis to “Amount of Fruit”:

```

<mx:AreaChart ...>
    ...
    <mx:horizontalAxisRenderer>
        <mx:AxisRenderer title="Amount of Fruit" />
    </mx:horizontalAxisRenderer>
</mx:AreaChart>

```

To set a style for the axis title, you can use the `axisTitleStyle` property of the chart control. The following example defines an `axisTitleStyle`:

```

<mx:Style>
    .myStyle {
        font-family: Verdana;
        font-size: 12;
        color: 0x4691E1;
        font-weight: bold;
        font-style: italic;
    }
</mx:Style>
<mx:ColumnChart axisTitleStyle="myStyle" dataProvider="{expenses}">
    <mx:horizontalAxis>
        <mx:horizontalAxisRenderer>
            <mx:AxisRenderer title="Expenses" />
        </mx:horizontalAxisRenderer>
    </mx:horizontalAxis>
    ...
</mx:ColumnChart>

```

To add a title to the vertical axis, you must embed the font used for the title style because the vertical axis title is rotated 90 degrees. The following example embeds the font and sets the style for the vertical axis's title:

```

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

```



```

<mx:ColumnChart axisTitleStyle="myEmbeddedStyle" dataProvider="{expenses}">
  <mx:horizontalAxis>
    ...
  <mx:horizontalAxis>
  <mx:horizontalAxisRenderer>
    <mx:AxisRenderer title="Expenses" />
  </mx:horizontalAxisRenderer>
  ...
</mx:ColumnChart>

```

You can take advantage of the fact that the chart applies the `axisTitleStyle` property without explicitly specifying it, as the following example shows:

```

<mx:Style>
  @font-face {
    font-family: myTitleFont;
    src: local("Arial");
  }
  .axisTitles {
    font-family: myTitleFont;
  }
  ColumnChart {
    axisTitleStyle: axisTitles;
  }
</mx:Style>

```

For information on embedding fonts, see [“Using embedded fonts” on page 423](#).

Defining axis labels

You use the `horizontalAxis` or `verticalAxis` to define the values of axis labels. But you can customize the value of axis labels using the `labelFunction` callback function on the `AxisRenderer`. To modify the label text, you use the `labelFunction` property of the `AxisRenderer` to specify a callback function. The function specified in `labelFunction` returns a string that Flex displays as the axis label.

The following is the required format of the callback function:

```
function function_name ( categoryName )
```

The following example defines a `labelFunction` for the `horizontalAxisRenderer`. In that function, Flex appends '04 to the axis labels, so that it displays the labels as Jan '04, Feb '04, and Mar '04:

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:Script>
    var expenses = [{Month: "Jan", Profit: 2000, Expenses: 1500 },
                    {Month: "Feb", Profit: 1000, Expenses: 200},
                    {Month: "Mar", Profit: 1500, Expenses: 500}];
    function defineLabel(catName) {
      return catName + " '04";
    }
  </mx:Script>
  <mx:Panel>
    <mx:ColumnChart id="column" dataProvider="{expenses}">

```

```

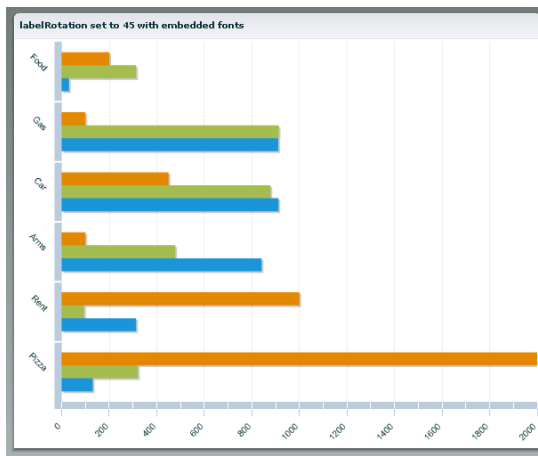
<mx:horizontalAxis>
    <mx:CategoryAxis dataProvider="{expenses}" categoryField="Month" />
</mx:horizontalAxis>
<mx:horizontalAxisRenderer>
    <mx:AxisRenderer title="Expenses" labelFunction="defineLabel"/>
</mx:horizontalAxisRenderer>
<mx:series>
    <mx:Array>
        <mx:ColumnSeries xField="Month" yField="Profit" name="Profit" />
        <mx:ColumnSeries xField="Month" yField="Expenses" name="Expenses" />
    </mx:Array>
</mx:series>
</mx:ColumnChart>
</mx:Panel>
</mx:Application>

```

Rotating axis elements

You can rotate axis labels using the `labelRotation` attribute of the `<mx:AxisRenderer>` tag. You specify a number from 1 to 180, in degrees. If you specify a negative number or set `labelRotation` to null, Flex determines an optimal angle and renders the labels.

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



To rotate text, you must embed the font in the Flex application. The following `<mx:Style>` block embeds the Arial font in the Flex application, and then applies that font to the chart control that rotates its labels 90 degrees:

```

<mx:Style>
    @font-face {
        font-family: labelFont;
        src: local("Arial");
    }
    BarChart{
        font-family: labelFont;
    }
</mx:Style>

```

```

<mx:BarChart dataProvider="{expenses}" showDataTips="true">
  <mx:horizontalAxisRenderer >
    <mx:AxisRenderer labelRotation="90"/>
  </mx:horizontalAxisRenderer>
  <mx:verticalAxisRenderer>
    <mx:AxisRenderer labelRotation="90"/>
  </mx:verticalAxisRenderer>
  ...
</mx:BarChart>

```

Formatting tick marks

Each chart, except for a pie chart, has an x-axis and a y-axis. Along each axis, Flex draws small markers, called *tick marks*, with a corresponding label. That label can be a text string or a numeric value.

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

You use the `tickPlacement` and `minorTickPlacement` properties of the `<mx:AxisRenderer>` tag to determine whether Flex displays tick marks and where Flex displays those tick marks.

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

| Value | Description |
|----------------------|---|
| <code>none</code> | Hides tick marks. |
| <code>inside</code> | Places tick marks on the inside of the axis line. |
| <code>outside</code> | Places tick marks on the outside of the axis line. This is the default value for the <code>tickPlacement</code> property. |
| <code>cross</code> | Places tick marks across the axis. |

The default value of `tickPlacement` is `outside`. The default value of `minorTickPlacement` is `none`.

You can align the tick marks with labels 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 appearance of tick marks on the chart's axes:

| Property | Description |
|---------------------------------|---|
| <code>tickLength</code> | The length, in pixels, of the major tick mark from the axis. |
| <code>minorTickLength</code> | The length, in pixels, of the minor tick mark from the axis. |
| <code>minorTickDivisions</code> | The number of divisions a segment between major tick marks is divided into. |

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

```
<mx:Style>
    .myAxisStyle {
        tickLength: 12;
        tickPlacement: inside;
        minorTickPlacement: none;
    }
</mx:Style>
<mx:ColumnChart dataProvider="{dataSet}" >
    <mx:horizontalAxisRenderer>
        <mx:AxisRenderer styleName="myAxisStyle"/>
    </mx:horizontalAxisRenderer>
    ...
</mx:ColumnChart>
```

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

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` to `false`. The default is `true`. The following example sets `showLine` to `false`:

```
<mx:AxisRenderer showLine="false" />
```

You can also apply `showLine` 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, as the following example shows:

```
<mx:horizontalAxisRenderer>
    <mx:AxisRenderer>
        <mx:axisStroke>
            <mx:Stroke color="0x884422" weight="2" alpha="75" />
        </mx:axisStroke>
    </mx:AxisRenderer>
</mx:horizontalAxisRenderer>
```

For more information about strokes, see [“Setting stroke styles” on page 508](#).

Setting stroke styles

You use the `<mx:Stroke>` tag with the chart series and chart renderer to control the properties of the lines that Flex uses to draw chart elements.

You can instantiate a `Stroke` in `ActionScript` using the `mx.graphics.Stroke` class, as the following example shows:

```
<mx:Script><![CDATA[
    import mx.graphics.Stroke;
```

```

...
var stroke : Stroke = new Stroke(0x001100);
...
]]></mx:Script>

```

The `<mx:Stroke>` tag has the following properties that you use to control the appearance of lines:

- `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. Possible values are `0` (invisible) through `100` (opaque). The default value is `100`.

Each chart series and chart renderer has one or more properties, such as the `stroke` property of the `BarSeries` or the `radialStroke` property of the `PieSeries`, that you use with the `<mx:Stroke>` tag to control line properties. The following example sets the line properties for a `BarChart` control:

```

<mx:BarChart id="chart" dataProvider="{dataSet}" >
  <mx:series>
    <mx:Array>
      <mx:BarSeries>
        <mx:xField>Sales</mx:xField>
        <mx:stroke>
          <mx:Stroke color="0x808080" weight="2" alpha="80" />
        </mx:stroke>
      </mx:BarSeries>
      <mx:BarSeries >
        <mx:xField>Revenue</mx:xField>
        <mx:stroke>
          <mx:Stroke color="0xC0C0C0" weight="2" alpha="80" />
        </mx:stroke>
      </mx:BarSeries>
    </mx:Array>
  </mx:series>
</mx:BarChart>

```

For each series, you define a line width of 2 pixels, one with a dark gray border (`0x808080`) and the other with a light gray border (`0xC0C0C0`).

Showing DataTips

You use the `showDataTips` property of chart controls to enable `DataTips`. `DataTips` are similar to `Flex ToolTips` in that they cause a small pop-up window to appear showing the data value for the data point under the mouse pointer.

Note: `DataTip` controls and `PieChart` labels are not the same, although they often show the same information. `PieChart` labels are always visible regardless of the location of the user's mouse pointer.

To enable DataTips, set the value of the chart control's `showDataTips` property to `true`, as the following example shows:

```
<mx:AreaChart id="chart" showDataTips="true" >
  ...
</mx:AreaChart>
```

The format of the DataTip depends on the chart type, but typically it displays the names of the fields in the data provider that supply the data at the selected location in the chart and the values of the data from the data provider.

To make the information in the DataTip understandable to users, you can define the series of your chart to have easily understood names.

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

```
<mx:BarSeries xField="f1" name="Sales" />
<mx:BarSeries xField="f2" name="Revenue" />
<mx:BarSeries xField="f3" name="Forecast" />
```

You can also name the axis to display labels in the DataTip. When the axis has a name, this name appears in the DataTip in italics before the label data. The following example names the axis to State:

```
<mx:verticalAxis>
  <mx:CategoryAxis name="State" dataProvider="{dataSet}" />
</mx:verticalAxis>
```

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"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:Script>
    var expenses = [{Expense: "Taxes", Amount: 2000},
                    {Expense: "Gas", Amount: 100},
                    {Expense: "Food", Amount: 200} ];
  </mx:Script>
  <mx:BarChart dataProvider="{expenses}" showDataTips="true">
    <mx:verticalAxis>
      <mx:CategoryAxis dataProvider="{expenses}" categoryField="Expense"
        name="Expense"/>
    </mx:verticalAxis>
    <mx:horizontalAxis>
      <mx:LinearAxis name="Amount"/>
    </mx:horizontalAxis>
    <mx:series>
      <mx:Array>
        <mx:BarSeries xField="Amount" yField="Expense" name="MY EXPENSES"/>
      </mx:Array>
    </mx:series>
  </mx:BarChart>
</mx:Application>
```

Customizing DataTip values

You can customize the text in DataTips 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 the event object that contains the `hitData` property, of type `mx.charts.HitData`. The `item` property of `hitData` contains a copy of the `dataProvider` object for the data point.

The following example defines a new callback function, `dtFunc`, that returns a formatted value for the DataTip:

```
<mx:Script><![CDATA[
    function dtFunc(eventObject:Object){
        return "The radius is " + eventObject.hitData.item.F3;
    }
]]></mx:Script>
<mx:BubbleChart id="chart"
    showDataTips="true" dataProvider="{dataSet}" dataTipFunction="dtFunc">
    <mx:series>
        <mx:Array>
            <mx:BubbleSeries xField="F1" yField="F2" radiusField="F3" />
        </mx:Array>
    </mx:series>
</mx:BubbleChart>
```

The event object inside a `dataTipFunction` callback function also gives you access to the series object that triggered the DataTip. The following example builds a more complex DataTip for a `dataTipFunction` callback function using the `element` property that represents the series:

```
function buildDataTip(eventObj) {
    // Get the hit data structure from the event object:
    var hd : mx.charts.HitData = eventObj.hitData;
    // Cast element to a columnSeries:
    var col : mx.charts.series.ColumnSeries =
        mx.charts.series.ColumnSeries(hd.element);
    // Get the yValue out of the data point:
    return col.name + "\n" + hd.data[col.yField];
}
```

The result is a two-line DataTip, with the series name and value of the data in the `y` series. The `HitData` object is also used when interacting with charting events. For more information about the `HitData` object, see [“About the HitData object” on page 527](#).

Your DataTip function can return HTML for formatted DataTips. Supported tags include ``, `<I>`, and `
`.

You can use the `mouseSensitivity` property to configure the distance, in pixels, around the mouse pointer where Flex reacts to data points. With this property, you can trigger DataTips to appear when the user moves the mouse pointer near a data point rather than onto the data point. For more information, see [“Changing mouse sensitivity” on page 529](#).

Using minField

Some series types let you specify a minimum value for the elements drawn on the screen. For example, in an `ColumnChart` control, you can have a second value specify the bottom of the column. To specify a bottom (or minimum) for the column, set the value of the series object's `minField` property to the data provider field.

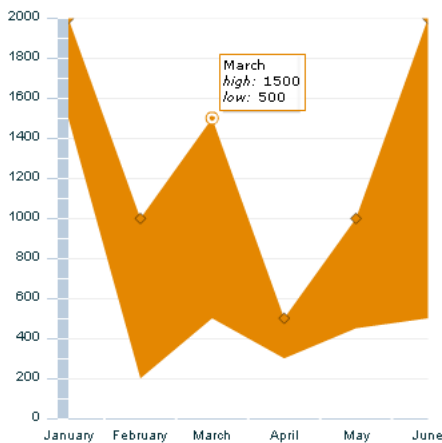
You can specify the `minField` property for the following chart series:

- `AreaSeries`
- `BarSeries`
- `ColumnSeries`

Setting a `minField` value creates two values on the axis for each column; for example:

```
<ColumnSeries name="Soda Sales" yField="F0" minField="F1">
```

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:



Using images in charts

To improve the appearance of a chart, you can import images using the `AssetRenderer`. The `AssetRenderer` lets you reference an image or SWF file for use in the chart, and scales the image based on the corresponding data value. Supported image types include JPEG, GIF, PNG, and SWF files.

To use the `AssetRenderer`, you use the `<mx:AssetRenderer>` child tag of the `<mx:renderer>` tag inside a data series.

The following example uses an `AssetRenderer` to import a tree icon into a column chart to represent each column in the series. The image replaces the default `SimpleBoxRenderer` class:

```
<mx:ColumnChart id="chart" dataProvider="{dataSet}" >
  <mx:series>
    <mx:Array>
      <mx:ColumnSeries yField="Newton" >
```

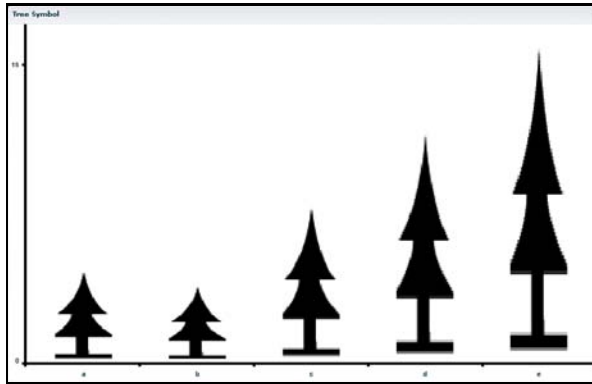


```

<mx:renderer>
    <mx:AssetRenderer source="@Embed('tree.jpg')" />
</mx:renderer>
</mx:ColumnSeries>
</mx:Array>
</mx:series>
</mx:ColumnChart>

```

In this example, Flex scales the imported tree image based on the data associated with each data point in the chart, just as it would scale the columns of a column chart. This means that data points with large values have a correspondingly large image, and data points with small values have a correspondingly small image. The following figure shows this chart:



If you import a SWF file, Flex resizes and plays the movie within the confines of the image area. As with other import functions in Flex, you can also import a symbol from a SWF file using the `@Embed('swf_name#xsymbol_name')` syntax.

If you import an image type that supports transparency, Flex maintains this transparency when you use the image in the chart control.

You can use the `AssetRenderer` for chart elements whose default renderers derive from the `BoxRenderer` interface. The following is a list of these renderers:

- `CircleRenderer`
- `CrossRenderer`
- `DiamondRenderer`
- `ShadowBoxRenderer`
- `SimpleBoxRenderer`
- `TriangleRenderer`

You can also add images with an `AssetRenderer` to the data points in a `LineChart` control, as described in the following section.

Adding images to LineChart controls

Line and area charts draw a line between each data point in a chart. By default, line charts use the `SimpleLineRenderer` and area charts use the `SimpleAreaRenderer` to draw those lines. For both chart types, you can also use the `pointRenderer` property to specify a second renderer for the same data series. This lets you specify one renderer to draw the line between each data point, and one renderer to draw a chart element at each data point.

With the `pointRenderer` property, you can specify any renderer that implements the `BoxRenderer` interface, such as the `CircleRenderer` or `CrossRenderer`. You can also use the `AssetRenderer` to import an image that is displayed at each data point.

The following example creates a line chart with two lines. One line uses the `<mx:AssetRenderer>` tag to import images for the data points, and the other uses a `CrossRenderer` class to draw crosses at each data point.

```
<mx:LineChart id="chart" dataProvider="{dataSet}" >
  <mx:series>
    <mx:Array>
      <mx:LineSeries yField="SanFrancisco" >
        <mx:stroke>
          <mx:Stroke color="0x3F5FC4" weight="3" alpha="100" />
        </mx:stroke>
        <!-- You can also set pointRadius directly on the LineSeries tag -->
        <mx:pointRadius>12</mx:pointRadius>
        <mx:pointRenderer>
          <mx:AssetRenderer source="@Embed('img/yellowclock.gif')" />
        </mx:pointRenderer>
      </mx:LineSeries>
      <mx:LineSeries yField="Newton" >
        <mx:stroke>
          <mx:Stroke color="0x3F5FC4" weight="3" alpha="100" />
        </mx:stroke>
        <!-- You can also set pointRadius directly on the LineSeries tag -->
        <mx:pointRadius>12</mx:pointRadius>
        <mx:pointRenderer>
          <mx:CrossRenderer />
        </mx:pointRenderer>
      </mx:LineSeries>
    </mx:Array>
  </mx:series>
</mx:LineChart>
```

Flex does not scale the icons defined by the `<mx:pointRenderer>` tag based on the value of the data point but uses the `pointRadius` property to specify a pixel value for the icon size.

Using fills

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. The fill lets you specify a pattern that defines how Flex draws the chart element. You can also use fills to specify background colors of the chart. Fills can be solid or use linear and radial gradients.

You use the `fill` property of the chart series to define the characteristics of the fill. The `LineSeries`, `LineRenderer`, and `AssetRenderer` are not affected by the `fill` property's settings. `PieSeries` support an array of `fills` rather than a single `fill` property.

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

```
<mx:ColumnChart id="chart" dataProvider="{dataSet}" >
  <mx:series>
    <mx:Array>
      <mx:ColumnSeries yField="Newton" >
        <mx:fill>
          <mx:SolidColor color="0x336699" />
        </mx:fill>
      </mx:ColumnSeries>
      <mx:ColumnSeries yField="San Francisco" >
        <mx:fill>
          <mx:SolidColor color="0xFF99FF"/>
        </mx:fill>
      </mx:ColumnSeries>
    </mx:Array>
  </mx:series>
</mx:ColumnChart>
```

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

With the `PieSeries`, you can use an 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:

```
<mx:PieChart id="chart" dataProvider="{expenses}" showDataTips="true">
  <mx:series>
    <mx:Array>
      <mx:PieSeries field="Amount" name="Amount" nameField="Expense">
        <mx:fills>
          <mx:Array>
            <mx:SolidColor color="0xFF0000" alpha="20" />
            <mx:SolidColor color="0x0000FF" alpha="20" />
            ...
          </mx:Array>
        </mx:fills>
      </mx:PieSeries>
    </mx:Array>
  </mx:series>
</mx:PieChart>
```

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"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:Script>
    var expenses = [{Expense: "Taxes", Amount: 2000},
                    {Expense: "Rent", Amount: 1000},
                    {Expense: "Bills", Amount: 100} ];
  </mx:Script>
  <mx:Panel title="fill set to red">
    <mx:BarChart dataProvider="{expenses}" showDataTips="true">
      <mx:verticalAxis>
        <mx:CategoryAxis dataProvider="{expenses}" categoryField="Expense" />
      </mx:verticalAxis>
      <mx:fill>
        <mx:SolidColor color="0x224466" />
      </mx:fill>
      <mx:series>
        <mx:Array>
          <mx:BarSeries xField="Amount"/>
        </mx:Array>
      </mx:series>
    </mx:BarChart>
  </mx:Panel>
</mx:Application>
```

Setting fills with CSS

You can use the `fill` property in an `<mx:Style>` declaration using CSS syntax. The following example sets the fill of the custom `myBarChartStyle` to `0xFF0000`:

```
<mx:Style>
  .myBarChartStyle {
    fill: 0xFF0000;
  }
</mx:Style>
```

Flex converts this to a solid color object.

In your MXML, you point to the custom style on the chart series, as the following example shows:

```
<mx:BarSeries styleName="myBarChartStyle" yField="Apples" name="Apples" />
```

Using a gradient fill

Flex includes two fill tags that let you specify a gradient fill, in which a *gradient* specifies a gradual color transition in the fill color. You use either the `<mx:LinearGradient>` or the `<mx:RadialGradient>` tag, along with the `<mx:GradientEntry>` child tag to specify a gradient fill:

- `<mx:LinearGradient>` Defines a gradient fill starting at a boundary of the chart element. You specify an array of `GradientEntry` objects to control gradient transitions.
- `<mx: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.
- `<mx: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. Possible values are 0 (invisible) through 100 (opaque). The default value is 100.
 - `ratio` Specifies where in the chart, as a percentage, Flex starts the transition to the next color. For example, if you set `ratio` 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.

The following example uses an `<mx:LinearGradient>` tag with three colors for a gradient fill of the chart's data series:

```
<mx:ColumnSeries yField="Newton" >
  <mx:fill>
    <mx:LinearGradient >
      <mx:entries>
        <mx:Array>
          <mx:GradientEntry color="0xC5C551" ratio="0" alpha="100" />
          <mx:GradientEntry color="0xFEFE24" ratio="33" alpha="100" />
          <mx:GradientEntry color="0xECEC21" ratio="66" alpha="100" />
        </mx:Array>
      </mx:entries>
    </mx:LinearGradient >
  </mx:fill>
</mx:ColumnSeries>
```

The `<mx:LinearGradient>` tag takes a single attribute, `angle`. By default, the `<mx:LinearGradient>` tag 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.

```
<mx:ColumnSeries yField="Newton" >
  <mx:fill>
    <mx:LinearGradient angle="90" >
      <mx:entries>
        <mx:Array>
```

```

        <mx:GradientEntry color="0xC5C551" ratio="0" alpha="100" />
        <mx:GradientEntry color="0xFEFE24" ratio="33" alpha="100" />
        <mx:GradientEntry color="0xECEC21" ratio="66" alpha="100" />
    </mx:Array>
</mx:entries>
</mx:LinearGradient>
</mx:fill>
</mx:ColumnSeries>

```

Using different alpha values with a fill

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. 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 possible values for the alpha property are 0 (invisible) through 100 (opaque).

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

```

<mx:AreaChart id="c1" dataProvider="{dataSet}" >
    <mx:series>
        <mx:Array>
            <mx:AreaSeries yField="Sales" >
                <mx:stroke>
                    <mx:Stroke color="0x9A9A00" weight="2" />
                </mx:stroke>
                <mx:fill>
                    <mx:SolidColor color="0x7EAEFF" alpha="30"/>
                </mx:fill>
            </mx:AreaSeries>
            <mx:AreaSeries yField="Revenue" >
                <mx:stroke>
                    <mx:Stroke color="0x9A9A00" weight="2" />
                </mx:stroke>
                <mx:fill>
                    <mx:SolidColor color="0xAA0000" alpha="30" />
                </mx:fill>
            </mx:AreaSeries>
        </mx:Array>
    </mx:series>
</mx:AreaChart>

```

If you define a gradient fill, you can set the alpha property as part of the array of

<mx:GradientEntry> tags, as the following example shows:

```

<mx:ColumnSeries yField="Newton" >
    <mx:fill>
        <mx:LinearGradient angle="90">
            <mx:entries>
                <mx:Array>
                    <mx:GradientEntry color="0xC5C551" ratio="0" alpha="100" />
                    <mx:GradientEntry color="0xFEFE24" ratio="33" alpha="100" />

```

```

        <mx:GradientEntry color="0xECEC21" ratio="66" alpha="0" />
    </mx:Array>
</mx:entries>
</mx:LinearGradient >
</mx:fill>
</mx:ColumnSeries>

```

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

Adding grid lines

All charts except the `PieChart` control have grid lines by default. Those grid lines can be controlled by the CSS `gridLinesStyle` property or with the chart series' `backgroundElements` property.

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

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.

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

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

For more information, see the `GridLines` class entry in *Flex ActionScript and MXML API Reference*.

Styling grid lines with MXML

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

```
<mx:LineChart id="chart" dataProvider="{dataSet}" >
    ...
    <mx:backgroundElements>
        <mx:Array>
            <mx:GridLines changeCount="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:backgroundElements>
</mx:LineChart>
```

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.

Styling grid lines with CSS

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

```
<mx:Style>
    .myStyle {
        direction: "both";
        horizontalShowOrigin: true;
        horizontalTickAligned: false;
        horizontalChangeCount: 3;
        verticalShowOrigin: false;
        verticalTickAligned: true;
        verticalChangeCount: 2;
        horizontalFill: 0xFFFFF0;
        horizontalAlternateFill: 0xE0FFFF;
    }
</mx:Style>
<mx:BubbleChart dataProvider="{expenses}" >
    <mx:horizontalAxis>
        <mx:LinearAxis minimum="-500" maximum="2000" />
    </mx:horizontalAxis>
    <mx:series>
        <mx:Array>
            <mx:BubbleSeries xField="Profit" yField="Expenses" radiusField="Amount"
                name="Bubble Series">
```



```

        <mx:fill>
          <mx:SolidColor color="#DDA0DD" />
        </mx:fill>
      </mx:BubbleSeries>
    </mx:Array>
  </mx:series>
</mx:backgroundElements>
<mx:Array>
  <mx:GridLines styleName="myStyle">
    <mx:verticalFill>
      <mx:SolidColor color="0xDDA0DD" alpha="20" />
    </mx:verticalFill>
    <mx:verticalAlternateFill>
      <mx:SolidColor color="0x98FB98" alpha="20" />
    </mx:verticalAlternateFill>
  </mx:GridLines>
</mx:Array>
</mx:backgroundElements>
</mx:BubbleChart>

```

Using Legend controls

Legend controls match the fill patterns on your chart to labels that describe the data series shown with those fills. Each entry in a Legend is known as a *Legend item*. A Legend item is made up of two basic parts, the marker and the label. The following image shows these parts:



This section describes how to add Legend controls to your charts and how to format the marker and label in the Legend controls.

Adding a Legend control to your chart

The `<mx:Legend>` tag lets you add a Legend control to your charts that displays the label for each data series in the chart and a key showing the chart element for the series.

You can use the following methods for adding Legend controls to your charts:

- [“Binding the Legend control to a chart identifier” on page 522](#)
- [“Manually defining LegendItem objects” on page 523](#)
- [“Creating a custom dataProvider for LegendItem objects” on page 524](#)

The following sections describe these methods.

Binding the Legend control to a chart identifier

The simplest way to create a Legend control is to pass a chart control to it using the `dataProvider` property, as the following example shows:

```
<mx:ColumnChart id="myChart" dataProvider="{Assets}">
    ...
</mx:ColumnChart>
<mx:Legend dataProvider="{myChart}" />
```

The Legend control creates the legend using information from the chart control. It matches the colors of the Legend markers to the fills of the data series. It also matches Legend labels with the names of the data series. If you do not name the data series, Flex does not add a marker label to the chart.

The autogeneration of Legend controls relies on elements of the charts being named. The following example shows each series with a `name` property. These entries make up the labels in the Legend:

```
<mx:Script>
    var expenses = [{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:BarChart id="chart" dataProvider="{expenses}" >
    <mx:verticalAxis>
        <mx:CategoryAxis dataProvider="{expenses}" categoryField="Expense" />
    </mx:verticalAxis>
    <mx:series>
        <mx:Array>
            <mx:BarSeries xField="Amount" name="The Amount" />
            <mx:BarSeries xField="Cost" name="The Cost" />
            <mx:BarSeries xField="Discount" name="The Discount" />
        </mx:Array>
    </mx:series>
</mx:BarChart>
<mx:Legend dataProvider="{chart}" />
```

A Legend for a PieChart control uses the `nameField` property of the data series to find values for the legend. The following example sets the `nameField` property of a PieChart control's data series to `Expense`. As a result, Flex uses the value of the `Expense` field in the data provider in the Legend.

```
<mx:Script>
    var expenses = [{Expense: "Taxes", Amount: 2000},
                    {Expense: "Rent", Amount: 1000},
                    {Expense: "Food", Amount: 200}];
</mx:Script>
<mx:Panel>
    <mx:PieChart id="pie" dataProvider="{expenses}" showDataTips="true">
        <mx:series>
            <mx:Array>
                <mx:PieSeries field="Amount" nameField="Expense"/>
            </mx:Array>
        </mx:series>
    </mx:PieChart>
</mx:Panel>
```

```

    </mx:PieChart>
    <mx:Legend dataProvider="{pie}" />
</mx:Panel>

```

The `nameField` property also defines the series for the `DataTips` and labels.

Manually defining LegendItem objects

You can define an array of `LegendItem` objects to control how Flex creates the Legend control. The `LegendItem` class is derived from the `UIObject` class and consists of a marker and a label.

You specify a marker by defining a fill and a stroke for that marker. The fill is the color of the marker and the stroke is the border around the marker. You define a label using the `label` attribute of the `<mx:LegendItem>` tag.

The following example defines two `LegendItem` objects, each with its own label and marker with a fill and a stroke:

```

<mx:Panel title="Legend">
  <mx:Legend>
    <mx:LegendItem label="Apples" fontWeight="bold" >
      <mx:fill>
        <mx:SolidColor color="green" />
      </mx:fill>
      <mx:stroke>
        <mx:Stroke color="0xFF0000" weight="1"/>
      </mx:stroke>
    </mx:LegendItem>
    <mx:LegendItem label="Oranges" fontWeight="bold" >
      <mx:fill>
        <mx:SolidColor color="red" />
      </mx:fill>
      <mx:stroke>
        <mx:Stroke color="0xFF0000" weight="1"/>
      </mx:stroke>
    </mx:LegendItem>
  </mx:Legend>
</mx:Panel>

```

When defining legend markers, you can use any type of valid fill; as a result, you can specify a `LinearGradient` for the marker definition, as the following example shows:

```

<mx:Legend>
  <mx:LegendItem label="Revenue" fontWeight="bold" >
    <mx:renderer>
      <mx:ShadowBoxRenderer/>
    </mx:renderer>
    <mx:fill>
      <mx:LinearGradient angle="90" >
        <mx:entries>
          <mx:Array>
            <mx:GradientEntry color="0xC5C551" ratio="0" alpha="100"/>
            <mx:GradientEntry color="0xFEFE24" ratio="33" alpha="100"/>
            <mx:GradientEntry color="0xECEC21" ratio="66" alpha="100"/>
          </mx:Array>
        </mx:entries>
      </mx:LinearGradient>
    </mx:fill>
  </mx:LegendItem>
</mx:Legend>

```

```

        </mx:LinearGradient>
    </mx:fill>
</mx:LegendItem>
</mx:Legend>

```

Creating a custom dataProvider for LegendItem objects

You can create a Legend control by passing an array of objects to the `dataProvider` attribute of the Legend control. The array consists of a set of objects that contain a label and a fill, and optionally a stroke. You can create the array using an `ActionScript` block and assign it to a variable that you later bind to the Legend with its `dataProvider` attribute.

The following example creates an array of objects called `countryList`. The example then binds the Legend control to the `countryList` array.

```

<mx:Script><![CDATA[
    import mx.graphics.SolidColor;
    import mx.graphics.Stroke;
    var countryList: Array;
    function initLegend() {
        var fill : SolidColor;
        var stroke : Stroke = new Stroke(0x001100);
        var label : String;
        var keyItems = new Array();
        var labels = ["Poland","Germany","England","Russia","Italy"];
        var colors = [0xFEE123,0xEE1103,0x12FC07,0x003322,0x0921E9];
        for (var i = 0; i < labels.length; i++) {
            fill = new SolidColor(colors[i]);
            keyItems[i]={label:labels[i],fill:fill,stroke:stroke};
        }
        countryList = keyItems;
    }
]}></mx:Script>
<mx:Legend dataProvider="{countryList}" initialize="initLegend()"/>

```

Formatting the Legend control

The Legend control is a subclass of the `mx.containers.Tile` class. As a result, you can use `Tile` properties and some `Container` properties to manipulate the Legend control. In addition, the Legend control has the following properties that you can use to format its appearance:

| Property | Type | Description |
|-----------------------------|--------|--|
| <code>labelPlacement</code> | String | Specifies the alignment of the LegendItem object's label. Valid values are <code>right</code> , <code>left</code> , <code>top</code> , and <code>bottom</code> . |
| <code>markerHeight</code> | Number | Specifies the height, in pixels, of the LegendItem object's marker. |
| <code>markerWidth</code> | Number | Specifies the width, in pixels, of the LegendItem object's marker. |

| Property | Type | Description |
|----------|--------|---|
| renderer | Object | Specifies a renderer for the LegendItem object's marker. The renderer must implement the BoxRenderer interface. For more information on the BoxRenderer interface, see "Using a renderer" on page 478 . |
| stroke | Object | Specifies the line stroke for the LegendItem object's marker. For more information on defining line strokes, see "Setting stroke styles" on page 508 . |

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

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

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

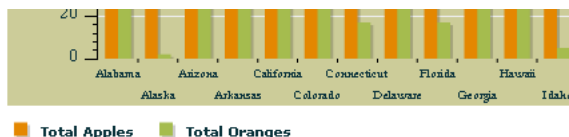
```
<mx:Panel title="Expenditures for FY04">
  <mx:PieChart ... >
    ...
  </mx:PieChart>
</mx:Panel>
<mx:Panel title="Legend">
  <mx:Legend ... />
</mx:Panel>
```

The direction property is a commonly used property inherited from the Tile container. This property of the <mx:Legend> tag causes the LegendItem objects to line up either horizontally or vertically. The default value of direction is vertical, in which case Flex stacks the LegendItem objects one on top of the other.

The following example sets the direction property to horizontal:

```
<mx:Legend dataProvider="{c1}" direction="horizontal" />
```

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



Stacking charts

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

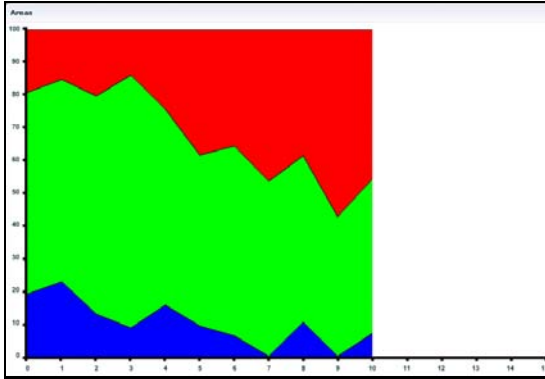
- `clustered` Chart elements for each series are grouped by category. This is the default value for `BarChart` and `ColumnChart` controls.
- `overlaid` Chart elements for each series are rendered on top of each other, with the element corresponding to the last series on top. This is the default value 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 percent that value contributes to the sum of the values for that category.

The following example creates a column chart that has three data series, stacked on top of each other:

```
<mx:ColumnChart id="chart" dataProvider="{dataSet}" type="stacked" >
  <mx:series>
    <mx:Array>
      <mx:ColumnSeries yField="Newton" >
        <mx:fill>
          <mx:SolidColor color="0x0000FF" />
        </mx:fill>
      </mx:ColumnSeries>
      <mx:ColumnSeries yField="New York" >
        <mx:fill>
          <mx:SolidColor color="0x00FF00" />
        </mx:fill>
      </mx:ColumnSeries>
      <mx:ColumnSeries yField="San Francisco" >
        <mx:fill>
          <mx:SolidColor color="0xFF0000" />
        </mx:fill>
      </mx:ColumnSeries>
    </mx:Array>
  </mx:series>
</mx:ColumnChart>
```

With an overlay, the last series appears on top, so it 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” on page 514](#).

A type property value of 100% configures the control to draw each series stacked on top of each other, adding up to 100% of the area. Each column represents the percent that that value contributes to the sum of the values for that category, as the following area chart shows:



Handling user interactions

The chart controls support the mouse events that are inherited from the `UIObject` class: `mouseMove`, `mouseover`, `mouseup`, `mousedown`, and `mouseout`. The base class for all chart controls, `ChartBase`, adds the following mouse events:

- `mouseoverData` Broadcast when the mouse pointer moves over a new data point.
- `mouseoutData` Broadcast when the closest data point under the mouse pointer changes.
- `mousemoveData` Broadcast when the mouse pointer moves while over a data point.
- `mouseClickData` Broadcast when the user clicks the mouse while over a data point.

Only data points within the radius determined by the `mouseSensitivity` property are considered. For more information, see [“Changing mouse sensitivity” on page 529](#).

About the HitData object

Flex generates an event object for each chart event. In addition to the standard `target` and `type` properties of the event object, Flex adds an additional property, `hitData`, to the event object. This property is an object instance of the `HitData` class and contains information about the data point that is closest to the mouse pointer at the time of the mouse event.

The following table describes the properties of the `HitData` object:

| Property | Description |
|-----------------------|---|
| <code>id</code> | A unique identifier that represents the data point. This is reserved for internal use. |
| <code>distance</code> | The distance between the data point on the screen and the point being queried. |
| <code>element</code> | The series object that rendered the data point. |
| <code>index</code> | The number of the data point in the <code>dataProvider</code> . |
| <code>item</code> | The data item from the <code>dataProvider</code> that the <code>HitData</code> structure describes. |

| Property | Description |
|----------|---|
| x | The x-coordinate of the data point on the screen. |
| y | The y-coordinate of the data point on the screen. |

The following example uses the `mouseClickData` event handler to display the data point value and screen coordinates for a column chart when the user clicks the mouse button. Because each column in the column chart is associated with a single data point value, clicking the mouse anywhere in the column displays the same information.

```
<mx:Script><![CDATA[
    // Define the event handler.
    function myClickEvent(e) {
        mySales.text="Sales= " + e.hitData.item.Sales;
        myXY.text="X/Y= " + e.hitData.x + "/" + e.hitData.y;
    }
]]></mx:Script>
<mx:Panel title="Event handling">
    <mx:ColumnChart id="chart" mouseClickData="myClickEvent(event)"
        dataProvider="{dataSet}" >
        <mx:series>
            <mx:Array>
                <mx:ColumnSeries yField="Sales" >
                    <mx:stroke>
                        <mx:Stroke color="0x0C0C0C" weight="1" />
                    </mx:stroke>
                    <mx:fill>
                        <mx:SolidColor color="0xC0C0C0" alpha="50" />
                    </mx:fill>
                </mx:ColumnSeries>
            </mx:Array>
        </mx:series>
    </mx:ColumnChart>

    <mx:HBox>
        <!-- Define TextArea controls for the event information. -->
        <mx:TextArea id="mySales" />
        <mx:TextArea id="myXY" />
    </mx:HBox>
</mx:Panel>
```

You can use the `element` property of the `HitData` structure to access properties of the data series that generated the event. The following example gets the value of the `yField` property of the `ColumnSeries`:

```
var fieldName = ColumnSeries(event.hitData.element).yField;
```


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. The nearest data point to the mouse pointer that is less than the number of pixels specified by the `mouseSensitivity` property is considered the current data point. The default value of the `mouseSensitivity` property is three pixels. If the mouse pointer is four or more pixels away from a data point, Flex does not populate the event object with a reference to the `HitData` object, because there is no associated data point. The `hitData` property in the event object is undefined.

The following example initially sets the `mouseSensitivity` property to 10, but lets the user change this value:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:Script>
    var expenses = [{ ... }];
    function changeSensitivity() {
      chart.mouseSensitivity = Number(myInput.text);
    }
  </mx:Script>
  <mx:PlotChart id="chart" dataProvider="{expenses}" showDataTips="true"
    mouseSensitivity="10">
    <mx:series>
      <mx:Array>
        <mx:PlotSeries xField="Profit" yField="Expenses" name="Ducats"/>
      </mx:Array>
    </mx:series>
  </mx:PlotChart>
  <mx:TextInput id="myInput" text="1"/>
  <mx:Button click="changeSensitivity()" label="Change Sensitivity" />
</mx:Application>
```

You can use the `mouseSensitivity` property to increase the area that triggers `DataTips` or emits chart-related events. If the mouse pointer is within the range of multiple data points, Flex chooses the closest data point.

Using the `findDataPoint()` method

You can use the chart control's `findDataPoint()` method to get a `HitData` object reference by passing in `x` and `y` coordinates. If the coordinates do not correspond to the location of a data point, the `findDataPoint()` method returns undefined. Otherwise, the `findDataPoint()` method returns a `HitData` object.

The following is the signature for the `findDataPoint()` method:

```
findDataPoint(x:Number, y:Number) : HitData
```

The following example creates a `PlotChart` control and records the location of the mouse pointer when the user clicks on the chart. It uses the `findDataPoint()` method to get a `HitData` object and then access the object's `index`, `x`, and `y` properties.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:Script>
```

```

var expenses = [{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}];

function handleClick(e) {
    // Get current coordinates from event object.
    var x = e.target.mouseX;
    var y = e.target.mouseY;
    // Use coordinates to get HitData object of current data point.
    var hd = chart.findDataPoint(x, y);
    if (hd != undefined) {
        ta.text = ta.text + "\n" + "data point " + hd.index + " found:
        (" + hd.x + ", " + hd.y + ")";
    } else {
        ta.text = ta.text + "\n" + "No data point found"
    }
}

function clearText() {
    ta.text = "";
}
</mx:Script>
<mx:PlotChart id="chart" mouseDown="handleClick(event)"
dataProvider="{expenses}" showDataTips="true" mouseSensitivity="5">
    <mx:series>
        <mx:Array>
            <mx:PlotSeries xField="Profit" yField="Expenses" />
        </mx:Array>
    </mx:series>
</mx:PlotChart>
<mx:TextArea id="ta" height="200" width="400" />
<mx:Button click="clearText()" label="Clear" />
</mx:Application>

```

Disabling interactivity

You can make the series in a chart ignore all mouse events by setting the `interactive` property to `false` on that series. The default is `true`. This lets you turn off mouse interactions for one series while allowing it for another.

Disabling the series interactivity causes the following results:

- The series does not show DataTips.
- The series does not generate a `hitData` structure on any mouse event.
- The series does not return a `hitData` structure when you call the `findDataPoint()` method on the chart.

The following example disables interactivity for the `PlotSeries`:

```

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

```

Using effects with charts

Chart controls support the standard Flex effects such as `Zoom` and `Fade`. In addition, chart data series have the following effects that apply to the data in the chart:

- `SeriesInterpolate`
- `SeriesSlide`
- `SeriesZoom`

Chart data series do not support the other Flex effects.

All chart controls and series support the standard Flex triggers and effects. You can use the following triggers on a chart control as tag properties:

- `creationCompleteEffect`
- `hideEffect`
- `mouseDownEffect`
- `mouseOut`
- `mouseOver`
- `mouseUpEffect`
- `moveEffect`
- `resizeEffect`
- `showEffect`

In addition to these triggers, Flex charts have their own unique triggers:

- `showDataEffect`
- `hideDataEffect`

This section briefly introduces you to using simple effects with chart controls. For more information, see [Chapter 18, “Using Behaviors,” on page 453](#).

The following sections describe how to use these effect triggers.

Using standard effect triggers

Chart controls support standard effects triggers such as `showEffect` and `hideEffect`.

The following example defines a set of wipes that Flex executes when the user toggles the chart's visibility with the Button control. In addition, Flex fades the chart in when it is created.

```
<mx:Effect>
  <mx:Sequence name="myWipes">
    <mx:WipeLeft />
    <mx:WipeRight />
    <mx:WipeUp />
    <mx:WipeDown />
  </mx:Sequence>
  <mx:Fade name="myFade" alphaFrom="0" alphaTo="100" duration="1000" />
</mx:Effect>
```

```

<mx:AreaChart id="area" dataProvider="{expenses}" showDataTips="true"
  hideEffect="myWipes" showEffect="myWipes" creationCompleteEffect="myFade">
  ...
</mx:AreaChart>
<mx:Button label="Toggle visibility" click="area.visible = !area.visible" />

```

Using charting effect triggers

Charts have two unique effect triggers, `showDataEffect` and `hideDataEffect`. You set these triggers on the data series for the chart. Whenever the data for a chart changes, Flex fires these triggers in succession.

The `hideDataEffect` trigger defines the effect that Flex uses as it hides the current data from view. The `showDataEffect` trigger defines the effect that Flex uses as it moves the current data into its final position on the screen.

Because Flex triggers the effects associated with `hideDataEffect` and `showDataEffect` when the data changes, there is “old” data and “new” data. The effect associated with the `hideDataEffect` trigger is the “old” data that will be replaced by the new data.

The order of events is as follows:

1. Flex first invokes the effect set with the `hideDataEffect` trigger for each element of a chart that is about to change. These triggers fire at the same time.
2. Flex then updates the chart with its new data. Any elements (including the gridlines and axes) that do not have effects associated with them update immediately to 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 into the chart.

Charting effects with data series

There are three effects that are unique to charting: `SeriesSlide`, `SeriesZoom`, and `SeriesInterpolate`. Each effect is used 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.

Each of the charting effects is a child tag of the `<mx:Effect>` tag, just as other effects such as `Zoom` or `Fade`. The following example shows the `SeriesSlide` effect and how it can be used to make the data slide in and out of a screen when the data changes:

```

<mx:Effect>
  <mx:SeriesSlide name="slideIn" duration="1000" direction="up"/>
  <mx:SeriesSlide name="slideOut" duration="1000" direction="down"/>
</mx:Effect>
<mx:ColumnChart>
  ...
  <mx:series>
    <mx:Array>
      <mx:ColumnSeries showDataEffect="slideIn" hideDataEffect="slideOut" />
    <mx:Array>
      <mx:series>

```

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 |
|-------------------------------------|---|
| <code>duration</code> | <p>The amount of time, in milliseconds, that Flex takes to complete the entire effect. This property defines the speed with which the effect executes.</p> <p>The <code>duration</code> property acts as a minimum duration. The effect can take longer based on the settings of other properties.</p> <p>The default is 500.</p> |
| <code>elementOffset</code> | <p>The amount of time, in milliseconds, that Flex delays the effect on each element in the series.</p> <p>Set <code>elementOffset</code> to 0 to affect all elements of the series at the same time. They start the effect at the same time and end it at the same time.</p> <p>Set <code>elementOffset</code> 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, then 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 will be longer.</p> <p>Set <code>elementOffset</code> to a negative value to have the effect begin from the last element and move backward through the list.</p> <p>The default is 20.</p> |
| <code>minimumElementDuration</code> | <p>The amount of time, in milliseconds, that an individual element should take to complete the effect.</p> <p>Charts with a variable number of data points in the series cannot reliably create smooth effects with only the <code>duration</code> property.</p> <p>For example, an effect with a duration of 1000 and <code>elementOffset</code> 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.</p> <p>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).</p> <p>With 10 elements, each element has only 100 milliseconds to complete the effect.</p> <p>The <code>minimumElementDuration</code> value 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 at least two of the following three properties are specified: <code>duration</code>, <code>offset</code>, and <code>minimumElementDuration</code>.</p> <p>The default is 0.</p> |
| <code>offset</code> | <p>The amount of time, in milliseconds, that Flex delays the start of the effect. Use this property to stagger effects on multiple series.</p> <p>The default is 0.</p> |

The following sections describe each of the charting effects in detail.

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 `SeriesSlide` as a `hideDataEffect`, the series slides from the current position onscreen to a position off of the screen, in the indicated direction. If you use `SeriesSlide` as a `showDataEffect`, the series slides from offscreen to a position onto the screen, in the indicated direction.

This tag has no additional parameters.

The following example creates an effect called `slideDown`. Each element starts its slide 30 milliseconds after the one before it, and takes at least 20 milliseconds to complete its slide. The entire effect takes at least one 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.

```
<mx:Effect>
  <mx:SeriesSlide duration="1000" direction="down"
    minimumElementDuration="20" elementOffset="30" name="slideDown" />
</mx:Effect>
<mx:AreaChart>
  ...
  <mx:series>
    <mx:Array>
      <mx:AreaSeries hideDataEffect="slideDown" showDataEffect="slideDown" />
    <mx:Array>
      <mx:series>
        </mx:series>
      </mx:series>
    </mx:series>
  </mx:AreaChart>
```

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 assigned as a `showDataEffect` or `hideDataEffect`.

The `SeriesZoom` effect can take several properties that define how the effect acts. The following table describes these properties:

| Property | Description |
|--|--|
| <code>horizontalFocus</code> <code>verticalFocus</code> | <p>Defines the location of the focal point of the zoom.</p> <p>You combine the <code>horizontalFocus</code> and <code>verticalFocus</code> properties to define where the data series zooms in and out from. For example, set <code>horizontalFocus</code> to <code>left</code> and <code>verticalFocus</code> to <code>top</code> to have the series data zoom to and from the top left corner of the either the element or the chart (depending on the setting of the <code>relativeTo</code> property).</p> <p>Valid values of <code>horizontalFocus</code> are <code>left</code>, <code>center</code>, <code>right</code>, and <code>undefined</code>.</p> <p>Valid values of <code>verticalFocus</code> are <code>top</code>, <code>center</code>, <code>bottom</code>, and <code>undefined</code>.</p> <p>If you specify only one of these two properties, then the focus is a horizontal or vertical line rather than a point. For example, when <code>horizontalFocus</code> is set to <code>left</code>, the element zooms to and from a vertical line along the left edge of its bounding box. Setting <code>verticalFocus</code> to <code>center</code> causes the element to zoom to and from a horizontal line along the middle of its bounding box.</p> <p>The default value for both properties is <code>center</code>.</p> |
| <code>relativeTo</code> | <p>Controls the bounding box used to calculate the focal point of the zooms. Valid values for <code>relativeTo</code> are <code>series</code> and <code>chart</code>.</p> <p>Set to <code>series</code> to zoom each element relative to itself. For example, each column of a <code>ColumnChart</code> zooms from the top left of the column.</p> <p>Set to <code>chart</code> to zoom each element relative to the chart area. For example, each column zooms from the top left of the axes, the center of the axes, etc.</p> |

The following example zooms in the data series from the top right of the chart. During the zoom in, Flex displays the last element in the series first because the `elementOffset` value is negative.

```
<mx:Effect>
  <mx:SeriesZoom duration="2000" minimumElementDuration="50"
    elementOffset="50" verticalFocus="top" horizontalFocus="left"
    relativeTo="chart" name="zoomOut" />
  <mx:SeriesZoom duration="2000" minimumElementDuration="50"
    elementOffset="-50" verticalFocus="top" horizontalFocus="right"
    relativeTo="chart" name="zoomIn" />
</mx:Effect>
<mx:AreaChart>
  ...
  <mx:series>
    <mx:Array>
      <mx:AreaSeries hideDataEffect="zoomOut" showDataEffect="zoomIn" />
    <mx:Array>
      <mx:series>
    </mx:series>
  </mx:AreaChart>
```

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 only use the `SeriesInterpolate` effect with the `showDataEffect` event trigger. It has no effect if set with a `hideDataEffect`.

This tag has no additional parameters.

The following example sets the `elementOffset` of `SeriesInterpolate` to 0. As a result, all elements move to their new locations simultaneously.

```
<mx:Effect>
  <mx:SeriesInterpolate duration="1000" minimumElementDuration="200"
    elementOffset="0" name="rearrangeData" />
</mx:Effect>
<mx:BubbleChart>
  ...
  <mx:series>
    <mx:Array>
      <mx:BubbleSeries showDataEffect="rearrangeData" />
    <mx:Array>
      <mx:series>
    </mx:series>
  </mx:BubbleChart>
```

Creating custom charts

The Flex charting controls include the `CartesianChart` control, a generic chart that is used as the base for all rectangular, two-dimensional charts. You can use this base chart class to create custom charts. You should have some understanding of how Flex charts work before using the `CartesianChart` control.

`CartesianChart` controls are subtly different from other chart controls. The `ColumnChart` control, for example, only lets you define its data series as a `ColumnSeries`. It also performs convenience functions for you based on that assumption. `CartesianChart` controls, on the other hand, can contain many different data series types and let you manually set properties that you would otherwise not be able to set.

Customizing bars and columns

You can customize the width of columns or bars in `ColumnChart` and `BarChart` controls by using a `CartesianChart` control in their place. If you create a column chart, Flex calculates the width of the columns and does not let you set them explicitly. (If you want to overlap columns or make them much thinner than their defaults, you use the `CartesianChart` and explicitly set the column properties.)

The following table describes style properties that you can use to manipulate the width of columns and bars in a CartesianChart:

| Property | Description |
|-----------------------------------|---|
| barWidthRatio columnWidthRatio | <p>Specifies how wide to render the columns or bars relative to the category width. A value of 1 uses the entire space, while a value of .6 uses 60% of the column or bar's available space. The ColumnChart and BarChart controls manage this value based on the value of the columnWidthRatio or barWidthRatio property.</p> <p>The actual column or bar width used is the smaller of what columnWidthRatio or barWidthRatio dictates and the maxColumnWidth or maxBarWidth property.</p> <p>The default value is .65.</p> |
| maxBarWidth maxColumnWidth | <p>Specifies the maximum width to draw the columns or bars, in pixels.</p> <p>The actual column or bar width used is the smaller of what maxColumnWidth or maxBarWidth dictates and the columnWidthRatio or barWidthRatio property. Clustered columns or bars divide this space proportionally among the columns or bars in each cluster.</p> <p>These properties do not have default values.</p> |
| offset | <p>Specifies how far to offset the center of the columns or bars from the center of the available space, relative to the category width.</p> <p>Set to 0 to center the columns or bars on the space.</p> <p>Set to 1 to shift the column or bar one entire category to the right (and -1 to shift it to the left).</p> <p>Set to -.5 to center the column or bar at the beginning of the available space.</p> <p>The ColumnChart and BarChart controls manage this value based on the value of the columnWidthRatio or barWidthRatio property.</p> <p>The default value is 0.</p> |

The following example defines a CartesianChart, populates it with ColumnSeries data, and sets the maxColumnWidth property to 20, which eliminates most of the whitespace between columns:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:Script>
    var expenses = [{Expense: "Taxes", Amount: 2000},
                    {Expense: "Rent", Amount: 1000},
                    {Expense: "Bills", Amount: 100},
                    {Expense: "Car", Amount: 450},
                    {Expense: "Gas", Amount: 100},
                    {Expense: "Food", Amount: 200}];
  </mx:Script>
  <mx:Panel>
    <mx:CartesianChart dataProvider="{expenses}" >
      <mx:horizontalAxis>
        <mx:CategoryAxis name="Expense" dataProvider="{expenses}"
          categoryField="Expense" />
      </mx:horizontalAxis>
      <mx:series>
        <mx:Array>
          <mx:ColumnSeries yField="Amount" maxColumnWidth="20"/>
        </mx:Array>
      </mx:series>
    </mx:CartesianChart>
  </mx:Panel>
</mx:Application>
```

```

        </mx:Array>
    </mx:series>
</mx:CartesianChart>
</mx:Panel>
</mx:Application>

```

Using multiple data series

The CartesianChart control lets you mix different data series in the same chart control. Using a CartesianChart control, you can create a column chart with a trend line running through it or mix any data series whose type derives from the CartesianChart control with any other, similar series.

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

- AreaSeries
- BarSeries
- BubbleSeries
- ColumnSeries
- LineSeries
- PlotSeries

The following example mixes a LineSeries and a ColumnSeries to create a column chart with a trend line:

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
    <mx:Script>
        var expenses = [{Expense: "Taxes", Amount: 2000, Profit:200},
                        {Expense: "Rent", Amount: 1000, Profit:100},
                        {Expense: "Bills", Amount: 100, Profit:10},
                        {Expense: "Car", Amount: 450, Profit:45},
                        {Expense: "Gas", Amount: 100, Profit:10},
                        {Expense: "Food", Amount: 200, Profit:20}];
    </mx:Script>
    <mx:Panel>
        <mx:CartesianChart dataProvider="{expenses}">
            <mx:horizontalAxis>
                <mx:CategoryAxis name="Expense" dataProvider="{expenses}"
                                categoryField="Expense" />
            </mx:horizontalAxis>
            <mx:series>
                <mx:Array>
                    <mx:ColumnSeries yField="Amount" />
                    <mx:LineSeries yField="Profit" />
                </mx:Array>
            </mx:series>
        </mx:CartesianChart>
    </mx:Panel>
</mx:Application>

```

CHAPTER 20

Using ToolTips

Macromedia Flex ToolTips are a flexible method of providing helpful information to your users. When a user moves the mouse pointer over a graphical component, the ToolTip pops up and text appears. You can use ToolTips to guide users through working with your application or customize them to provide additional functionality. This chapter describes how to use ToolTips in your Flex applications.

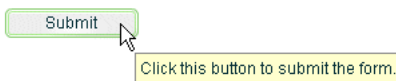
Contents

| | |
|---------------------------------|-----|
| About ToolTips | 539 |
| Creating ToolTips | 540 |
| Using the ToolTip Manager | 543 |

About ToolTips

ToolTips are a standard feature of many desktop applications. They make the application easier to use by displaying messages when the user rolls the mouse pointer over an on-screen element, such as a Button control.

The following figure shows ToolTip text that Flex shows when the user hovers the mouse pointer over a button:



When the user moves the mouse pointer away from the component or, in the case of the button, clicks the component, the ToolTip disappears. If the mouse pointer remains over the component, the ToolTip eventually disappears. Only one ToolTip is visible at a time.

You can set the time it takes for the ToolTip to appear when a user moves the mouse pointer over the component, and set the amount of time it takes for the ToolTip to disappear.

Child controls inherit their parent's ToolTips. As a result, if the ToolTip is defined on a container, all children of that container will have that ToolTip, unless they override it with a new ToolTip or assign an empty string for ToolTip text.

Flex ToolTips support style sheets and the dynamic loading of ToolTip text. ToolTip text does not support embedded HTML. For more information on using style sheets and dynamic loading of text, see [“Setting styles in ToolTips” on page 541](#) and [“Using dynamic ToolTip text” on page 546](#).

Creating ToolTips

Every visual Flex component supports a `tooltip` property. You set the value of the `tooltip` property to a text string and, when the mouse pointer hovers over that component, Flex displays the text string. The following example sets the `tooltip` property text for a Button control:

```
<mx:Button id="myButton" label="Submit" width="100" tooltip="Click this button
to submit the form." click="doSomething();" />
```

To set the value of a ToolTip in ActionScript, use the `tooltip` property of the component. The following example sets the `tooltip` property of a Button control:

```
<mx:Button id="myButton" label="Go" width="100" click="doSomething();" />
<mx:Script><![CDATA[
    function doSomething() {
        myButton.tooltip = "Click this button to Do Something";
    }
]></mx:Script>
```

Children of containers inherit the ToolTips of their parents. If you add a Button control to a Panel container that has a ToolTip, the user sees the Panel container’s ToolTip text when they move their mouse pointer over the Button control. A child can clear the ToolTip text of an ancestor by setting `tooltip=""`.

The following example shows the inheritance of ToolTip text and how to override it:

```
<mx:VBox tooltip="VBOX">
    <mx:Button id="b1" tooltip="BUTTON"/>
    <mx:Button id="b2" tooltip="" />
    <mx:Button id="b3"/>
</mx:VBox>
```

When the mouse pointer is over button b1, the ToolTip displays BUTTON. When the mouse pointer is over button b2, no ToolTip text appears. When the mouse is over button b3 or is over anywhere in the VBox container except the buttons, the ToolTip displays VBOX.

The tabs of the TabNavigator container use the ToolTips of their children. If you add a ToolTip to a child view of a TabNavigator container, the ToolTip appears when the mouse is over the tab for that view, but not when the mouse is over the view itself. ToolTips in the Accordion container also function this way.

There is no limit to the size of the ToolTip text, although long messages can be difficult to read. When the ToolTip text reaches the width of the ToolTip box, the text wraps to the next line. You can add line breaks in ToolTip text. In ActionScript, you use the `\n` escaped newline character. In MXML tags, you use the `` XML entity.

The following examples show using the `\n` escaped newline character and the `` entity:

```
<mx:Script><![CDATA[
    function doSomething() {
        // Use the \n to force a line break in ActionScript
        myButtonAS.toolTip = "Click this button \n to do something";
    }
]]></mx:Script>
<mx:Button id="myButtonAS" label="Do Something" width="100"
    click="doSomething();" />
<!-- Use &#13; to force a line break in MXML tags -->
<mx:Button id="myButton" label="Submit" width="100" toolTip="Click this button
    &#13; to submit the form." />
```

You also have some flexibility in formatting the text of the `ToolTip`. You can access the `mx.controls.ToolTip` class to apply styles and change other settings for all `ToolTip`s in your application. The following sections describe how to set styles on the `ToolTip` text and box.

Setting styles in ToolTips

You can change the appearance of `ToolTip` text and the `ToolTip` box using Cascading Style Sheets (CSS) syntax or the `mx.styles.StyleManager` class. Changes to `ToolTip` styles apply to all `ToolTip`s in the current application.

You use a type selector in the `<mx:Style>` tag to set the styles of your `ToolTip`s in CSS syntax. The following example sets the font characteristics of the type selector `ToolTip` using CSS syntax:

```
<mx:Style>
    ToolTip { font-family: "Arial"; font-size: 9; font-style: "italic"; font-
        weight: "bold"; color: "0xFAFAD2" }
</mx:Style>
```

To use the `StyleManager` class, apply a style to the `ToolTip` type selector, as the following example shows:

```
<mx:Script><![CDATA[
    import mx.styles.StyleManager;
    function setTTStyle() {
        StyleManager.styles.ToolTip.fontWeight = "bold";
    }
]]></mx:Script>
```

`ToolTip`s use inheritable styles that you set globally. For example, you can set the `fontWeight` of `ToolTip`s with the `StyleManager` by setting it on the global style sheet, as the following example shows:

```
StyleManager.styles.global.fontWeight = "bold";
```

The following table describes the ToolTip styles that you can change. If you set a style that is inheritable on the ToolTip's parent container, the ToolTip inherits that style.

| Property | Description | Inheritable |
|------------------------------|--|-------------|
| <code>backgroundColor</code> | The color of the ToolTip box, expressed as a hexadecimal value; for example, #FFFFFF. | No |
| <code>borderColor</code> | The color of the border around the ToolTip box, expressed as a hexadecimal value; for example, #CCCC66. | Yes |
| <code>borderStyle</code> | The bounding box style. The possible values are: <code>none</code> , <code>solid</code> , <code>inset</code> , and <code>outset</code> . The default value is <code>inset</code> . | No |
| <code>color</code> | The color of the font, expressed as a hexadecimal value; for example, #FFFFFF. | Yes |
| <code>fontFamily</code> | The name of the font face; for example, Times or Arial. | Yes |
| <code>fontSize</code> | The size, in points, of the font; for example, 12. | Yes |
| <code>fontStyle</code> | Determines whether the text is italic. Recognized values are <code>normal</code> and <code>italic</code> . The default value is <code>normal</code> . | Yes |
| <code>fontWeight</code> | Determines whether the text is bold. Recognized values are <code>normal</code> and <code>bold</code> . The default value is <code>normal</code> . | Yes |
| <code>marginBottom</code> | The number of pixels that ToolTip text is offset from the bottom of the ToolTip box. | No |
| <code>marginLeft</code> | The number of pixels that ToolTip text is offset from the left of the ToolTip box. | No |
| <code>marginRight</code> | The number of pixels that ToolTip text is offset from the right of the ToolTip box. | No |
| <code>marginTop</code> | The number of pixels that ToolTip text is offset from the top of the ToolTip box. | No |
| <code>shadowColor</code> | The color of the ToolTip box's shadow, expressed as a hexadecimal value; for example, #FFFFFF. | Yes |
| <code>textAlign</code> | Alignment of text within the ToolTip box. Recognized values are <code>left</code> , <code>right</code> , and <code>center</code> . The default value is <code>right</code> . | Yes |
| <code>textDecoration</code> | Determines whether the text is underlined. Recognized values are <code>none</code> and <code>underline</code> . The default value is <code>none</code> . | No |

For more information on using styles, see [Chapter 16, “Using Styles and Fonts,”](#) on page 395.

Setting ToolTip width

You can set the width of the ToolTip box by changing the `maxWidth` property of the `mx.controls.ToolTip` class. For example, the following line changes the maximum width of the ToolTip box to 100 pixels:

```
mx.controls.ToolTip.maxWidth = 100;
```

The `maxWidth` property specifies the maximum width in pixels for new `ToolTip` boxes. Flex wraps the text of a `ToolTip` onto multiple lines to ensure that the width does not exceed this value. If the text in the `ToolTip` box is not as wide as the `maxWidth` property, Flex creates a box only wide enough for the text to fit.

The default `maxWidth` value is 300. The minimum `maxWidth` value is 30. If the `maxWidth` value exceeds the width of the application, Flex clips the text in the `ToolTip` box.

ToolTip events

`ToolTip`s trigger the following events:

- **showToolTip** Broadcast when the `ToolTip` text box becomes visible.
- **hideToolTip** Broadcast when the `ToolTip`'s state changes from visible to invisible.

In addition to the `type` and `target` properties, the event object for `ToolTip` events references the `ToolTip` in its `toolTip` property. With a reference to the `ToolTip`, you can then access the `ToolTip` text with the `text` property.

The following example displays a message (including the original `ToolTip`'s text) in the `TextArea` control in response to the `showToolTip` event:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
  initialize="this_init()">
  <mx:Label id="myLabel" toolTip="ToolTip" text="Mouse Over Me"/>
  <mx:Script><![CDATA[
    private function this_init() {
      myLabel.addEventListener("showToolTip",mx.utils.Delegate.create
        (this,testHandler));
    }
    private function testHandler(event) {
      tal.text = "The showToolTip event was triggered" + newline + "text: " +
        event.toolTip.text;
    }
  ]]></mx:Script>
  <mx:TextArea id="tal" width="200" height="100" />
</mx:Application>
```

Using the ToolTip Manager

The `ToolTipManager` class lets you set basic `ToolTip` functionality, such as display delay and the disabling of `ToolTip`s. The `ToolTipManager` class also contains a reference to the current `ToolTip` in its `currentToolTip` property.

This section describes how to use the `ToolTip Manager`.

Enabling and disabling ToolTips

You can enable and disable `ToolTip`s in your Flex applications. When you disable `ToolTip`s, no `ToolTip` box appears when the user moves the mouse pointer over a visible component, regardless of whether that component's `toolTip` property is set.

You use the `enabled` property of the `ToolTipManager` to enable or disable ToolTips. You set this property to `true` to enable ToolTips or `false` to disable ToolTips. The default value is `true`.

The following example toggles ToolTips on and off when the user clicks the Toggle ToolTips button:

```
<mx:Script><![CDATA[
    function toggleTT() {
        if (mx.managers.ToolTipManager.enabled == false) {
            mx.managers.ToolTipManager.enabled = true;
        } else {
            mx.managers.ToolTipManager.enabled = false;
        }
    }
}]></mx:Script>
<mx:Button label="Toggle ToolTips" width="100" click="toggleTT();" />
```

Setting delay times

The *delay time* is a measurement of time that passes before something takes place. For example, after you move the mouse pointer over a component, there is a brief delay before the ToolTip appears. This gives someone who is not looking for ToolTip text enough time to move the mouse pointer away before seeing the pop-up.

The `ToolTipManager` lets you set the length of time that passes before the ToolTip box appears, and the length of time that a ToolTip remains on the screen when a mouse pointer hovers over the component.

You set the value of the `ToolTipManager` `showDelay` and `hideDelay` properties in your ActionScript code blocks. The following table describes the time delay properties of the `ToolTipManager`:

| Property | Description |
|------------------------|---|
| <code>showDelay</code> | <p>The length of time, in milliseconds, that Flex waits before displaying the ToolTip box when a user moves the mouse pointer over a component that has a ToolTip.</p> <p>To make the ToolTip appear instantly, set the <code>showDelay</code> property to 0. The default time is 500 milliseconds or half of a second.</p> |
| <code>hideDelay</code> | <p>The amount of time, in milliseconds, that Flex waits to hide the ToolTip box after it appears. After Flex hides a ToolTip box, the user must move the mouse pointer off the component and back onto it to see the ToolTip box again.</p> <p>If you set the <code>hideDelay</code> property to 0, Flex does not display the ToolTip. Macromedia recommends using the default time of 10,000 milliseconds, or 10 seconds.</p> <p>If you set the <code>hideDelay</code> property to <code>Infinity</code>, Flex does not hide the ToolTip until the user triggers an event (such as moving the mouse pointer off the component). The following example sets the <code>hideDelay</code> property to <code>Infinity</code>:</p> <pre>mx.managers.ToolTipManager.hideDelay = Infinity;</pre> |

The following example uses the Application control's `initialize` event to set the starting values for the `ToolTipManager`:

```
<mx:Application width='500' height='300' xmlns:mx="http://www.macromedia.com/
2003/mxml" initialize="initApp();" >
  <mx:Script><![CDATA[
    function initApp() {
      mx.managers.ToolTipManager.enabled = true; // Optional. Default is true.
      mx.managers.ToolTipManager.showDelay = 0; // Display immediately.
      mx.managers.ToolTipManager.hideDelay = 3000; // Hide after 3 seconds.
    }
  ]]></mx:Script>
  ...
</mx:Application>
```

Using effects with ToolTips

You can use a custom effect or one of the standard Flex effects with ToolTips. You set the `showEffect` property of the `ToolTipManager` to point to the effect that you want to be triggered whenever a Tooltip is displayed. You can only use one Tooltip effect in each application.

To use an effect with your ToolTips:

1. Name the effect and define its properties using the `<mx:Effect>` tag.
2. Set the `showEffect` property of the `ToolTipManager` to set that effect for your ToolTips:

```
mx.managers.ToolTipManager.showEffect = "MyCustomEffect";
```

For more information about using effects and defining custom effects, see [Chapter 18, “Using Behaviors,”](#) on page 453.

The following example uses the Fade effect so that ToolTips fade in when the user moves the mouse pointer over a component:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" width="600"
height="600" initialize="app_init();" >
  <mx:Script><![CDATA[
    function app_init() {
      mx.managers.ToolTipManager.showEffect = "ToolTipFadeIn";
    }
  ]]></mx:Script>
  <mx:Effect>
    <mx:Fade name="ToolTipFadeIn" alphaFrom="0" alphaTo="100"
duration="1000"/>
  </mx:Effect>
  <mx:Button label="Do Nothing" tooltip="Click me: nothing will happen."/>
</mx:Application>
```

To turn off the Tooltip effect, use the `ToolTipManager` `hideEffect` property; for example:

```
mx.managers.ToolTipManager.hideEffect = "ToolTipFadeIn";
```

Using dynamic ToolTip text

You can use ToolTips for more than just displaying static help text to the user. You can also bind the ToolTip text to data or component text. This lets you use ToolTips as a part of the user interface, showing drill-down information, query results, or more helpful text that is customized to the user experience.

You bind the value of the ToolTip text to the value of another control's text using curly braces ({ }).

The following example inserts the value of the txtTo TextInput control into the ToolTip text when the user moves the mouse pointer over the Button control:

```
<mx:TextInput id="txtTo" width="300" />
<mx:Button label="Send" tooltip="Send e-mail to {txtTo.text}"/>
```

In this example, if the user enters **fred@fred.com** in the TextInput box, and then moves the mouse pointer over the button, Flex displays the message “Send e-mail to fred@fred.com” in the ToolTip box.

CHAPTER 21

Using the Cursor Manager

The Macromedia Flex Cursor Manager lets you control the cursor image in your Flex application. You can use the Cursor Manager to provide visual feedback to users to indicate when to wait for processing to complete, to indicate allowable actions, or to provide other types of feedback. The cursor image can be a JPEG, GIF, PNG, or SVG image, or a SWF file.

Contents

| | |
|--------------------------------|-----|
| About the Cursor Manager | 547 |
| Cursor Manager syntax | 551 |

About the Cursor Manager

By default, Flex uses the system cursor as the application cursor. You control the system cursor by using the settings of your operating system.

The Flex Cursor Manager lets you control the cursor image in your Flex application. For example, if your application performs processing that requires the user to wait until the processing completes, you can change the cursor so that it reflects the waiting period. In this case, you can change the cursor to an hourglass or other image.

You also might want to change the cursor to provide feedback to the user to indicate the actions that the user can perform. For example, you can use one cursor image to indicate that user input is enabled, and another to indicate that input is disabled.

You can use a JPEG, GIF, PNG, or SVG image, or a SWF file as the cursor image.

Using the Cursor Manager

To use the Cursor Manager, you import the `mx.managers.CursorManager` class into your application, and then reference its properties and methods.

The Cursor Manager controls a prioritized list of cursors, where the cursor with the highest priority is currently visible. If the cursor list contains more than one cursor with the same priority, the Cursor Manager displays the most recently created cursor.

You create a new cursor, and set an optional priority for the cursor, using the `setCursor()` method of the `CursorManager` class. This method adds the new cursor to the cursor list. If the new cursor has the highest priority, it is displayed immediately. If the priority is lower than a cursor already in the list, it is not displayed until the cursor with the higher priority is removed.

To remove a cursor from the list, you use the `removeCursor()` method. If the cursor is the currently displayed cursor, the Cursor Manager displays the next cursor in the list, if one exists. If the list ever becomes empty, the Cursor Manager displays the default system cursor.

The `setCursor()` method has the following signature:

```
setCursor(cursorSymbol:String, priorityLevel:Number, xOffset:Number,
          yOffset:Number) : Number
```

The following table describes the arguments for the `setCursor()` method:

| Argument | Description | Req/Opt |
|----------------------------|--|----------|
| <code>cursorSymbol</code> | Specifies a String that contains the symbol name of the cursor to display. | Required |
| <code>priorityLevel</code> | Specifies a Number that contains the priority level of the cursor. Possible values are <code>CursorManager.HIGHPRIORITY</code> , <code>CursorManager.MEDIUMPRIORITY</code> , and <code>CursorManager.LOWPRIORITY</code> . The default value is <code>CursorManager.MEDIUMPRIORITY</code> . | Optional |
| <code>xOffset</code> | Specifies a Number that contains the x offset of the <code>cursorSymbol</code> relative to the mouse pointer. The default value is 0. | Optional |
| <code>yOffset</code> | Specifies a Number that contains the y offset of the <code>cursorSymbol</code> relative to the mouse pointer. The default value is 0. | Optional |

This method returns the ID of the new cursor. You pass the ID to the `removeCursor()` method to delete the cursor. This method has the following signature:

```
static removeCursor(cursorID:Number) : Void
```

Creating and removing a cursor

The following example changes the cursor to a *wait* cursor during the loading of a large image file. After the load completes, the application removes the wait cursor and returns the cursor to the system cursor.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

<mx:Script>
  <![CDATA[
    import mx.managers.CursorManager;

    // Define a variable to hold the cursor ID.
    var cursorID : Number = 0;

    // Embed the cursor symbol.
    [Embed(source="wait.jpg")]
    var waitCursorSymbol:String;
```

```

// Define event handler to display the wait cursor and to load the image.
function initImage(event)
{
    cursorID = CursorManager.setCursor(waitCursorSymbol);
    image1.load("DSC00034.JPG");
}

// Define an event handler to remove the wait cursor.
function hideCursor(event)
{
    CursorManager.removeCursor(cursorID);
}
]]>
</mx:Script>

<mx:VBox>
    <!-- Loader control to load the image. -->
    <mx:Loader id="image1" complete="hideCursor(event)" />

    <!-- Button triggers the load. -->
    <mx:Button id="myButton" label="Show" click="initImage(event)"/>

</mx:VBox>
</mx:Application>

```

This example uses a JPEG image as the cursor image. You can also use a SWF file, as the following example shows:

```

[Embed(source="wait.swf")]
var waitCursorSymbol:String;

```

Or, you can reference a symbol from a SWF file, as the following example shows:

```

[Embed(source="cursorList.swf" symbol="wait")]
var waitCursorSymbol:String;

```

An advantage to using a SWF file is that you can create an animated cursor.

Setting a busy cursor

Flex defines a default busy cursor that you can use to indicate to the user that your application is processing, and that they should wait until that processing completes before the application will respond to inputs. The default busy cursor is a rotating circle.

To support the busy cursor, the Cursor Manager defines the following two methods:

- **setBusyCursor()** Displays the busy cursor. The busy cursor has a priority of `CursorManager.LOWPRIORITY`. Therefore, if the cursor list contains a cursor with a higher priority, the busy cursor is not displayed until you remove the higher-priority cursor. If you want to create a busy cursor at a higher priority level, use the `setCursor()` method, as the following example shows:

```

CursorManager.setCursor(CursorManager.BUSYCURSORSYMBOL,
    CursorManager.HIGHPRIORITY);

```

- `removeBusyCursor()` Removes the busy cursor from the cursor list. If other busy cursor requests are still active in the cursor list, which means that you called the `setBusyCursor()` method more than once, a busy cursor does not disappear until you remove all busy cursors from the list.

You can modify the previous example to use the default busy cursor, as the following example shows:

```
<mx:Script>
  <![CDATA[
    import mx.managers.CursorManager;

    function initImage(event)
    {
      CursorManager.setBusyCursor();
      image1.load("DSC00034.JPG");
    }

    function hideCursor(event)
    {
      CursorManager.removeBusyCursor();
    }

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

You do not have to import a cursor symbol or declare a variable to track the cursor ID.

Setting the busy cursor does not prevent a user from interacting with your application; a user can still enter text or select buttons. However, all containers support the `enabled` property. By default, this property is set to `true` to enable user interaction with the container and with the container's children. If you set `enabled` to `false`, Flex dims the color of the container and of all of its children, and blocks user input to the container and to all of its children. To block user interaction, set the `enabled` property to `false` when you display a busy cursor.

Using the Cursor Manager with the Loader control and other tags

To support the Cursor Manager, Flex adds the `showBusyCursor` property to the Loader control, and to the `<mx:WebService>`, `<mx:HttpService>`, and `<mx:RemoteObject>` tags. The default value is `false` for the Loader control, and `true` for the `<mx:WebService>`, `<mx:HttpService>`, and `<mx:RemoteObject>` tags.

For the Loader control, if you set the `showBusyCursor` property to `true`, Flex displays the busy cursor when the first progress event of the control is triggered, and hides the busy cursor when the complete event is triggered. The following example shows how you can rewrite the example in the section [“Setting a busy cursor” on page 549](#):

```
<mx:VBox>
  <!-- Loader control to load the image. -->
  <mx:Loader id="image1" showBusyCursor="true" />

  <!-- Button triggers the load. -->
  <mx:Button id="myButton" label="Show" click="initImage()"/>

</mx:VBox>
```

Cursor Manager syntax

The following sections describe the syntax of the classes that make up the Cursor Manager.

Class `mx.managers.CursorManager` syntax

The following table describes the syntax of the `mx.managers.CursorManager` class:

| Property/Method | Type | Use | Description |
|---------------------------------|--------|----------|--|
| <code>currentCursorID</code> | Number | Property | Read-only property that returns the ID of the currently displayed cursor. |
| <code>removeBusyCursor()</code> | | Method | Removes the busy cursor from the cursor list. If other busy cursor requests are still active in the cursor list, which means that you called the <code>setBusyCursor()</code> method more than once, the busy cursor does not disappear until you remove all busy cursors from the list. |
| <code>removeCursor()</code> | | Method | Removes the specified cursor from the list of active cursors. If the cursor was the currently displayed cursor, this method displays the next cursor in the list, if one exists. This method returns nothing. |
| <code>setBusyCursor()</code> | | Method | Inserts the busy cursor into the cursor list. The busy cursor is given a priority of <code>CursorManager.LOWPRIORITY</code> . If no cursor with a higher priority is defined in the cursor list, the busy cursor is displayed. |
| <code>setCursor()</code> | | Method | Assigns a cursor in the list of active cursors. The cursor might not display immediately, depending on the number of other active cursors and the priority level of the cursors. This method returns the ID of the cursor. |

Syntax for controls that directly support the Cursor Manager

The Cursor Manager adds the following optional property to the Loader control, and to the `<mx:WebService>`, `<mx:HttpService>`, and `<mx:RemoteObject>` tags:

| Property | Type | Use | Description |
|-----------------------------|---------|----------|---|
| <code>showBusyCursor</code> | Boolean | Property | If set to <code>true</code> , specifies to display the busy cursor when the first <code>progress</code> event of the control is triggered, and hide the busy cursor when the <code>complete</code> event is triggered. The default value for the Loader control is <code>false</code> . The default value for the <code><mx:WebService></code> , <code><mx:HttpService></code> , and <code><mx:RemoteObject></code> tags is <code>true</code> . |

CHAPTER 22

Using the Drag and Drop Manager

The Drag and Drop Manager lets you move data from one place in a Macromedia Flex application to another. This feature is especially useful in a visual application where your data can be items in a list, images, or Flex components.

Contents

| | |
|---|-----|
| About the Drag and Drop Manager | 553 |
| Using a List, Tree, or DataGrid control | 562 |
| Drag and Drop Manager syntax | 565 |

About the Drag and Drop Manager

Visual development environments typically let you manipulate objects in an application by selecting them with a mouse and moving them around the screen. The Flex Drag and Drop Manager lets you select an object such as an item in a List control, or a Flex control such as an Image control, and then drag it over another component to add it to that component.

All Flex components support the drag-and-drop operation. In addition, Flex includes support for the drag-and-drop operation for the List, Tree, and DataGrid controls.

Using the drag-and-drop operation

A user initiates a drag-and-drop operation by using the mouse to select a Flex component, or an item in a Flex component, and then moving the component while holding down the mouse button. For example, a user selects an item in a List control with the mouse and, while holding down the mouse button, moves the mouse several pixels. The selected component is called the *drag initiator*.

While holding down the mouse button, the user moves the mouse around the Flex application. Flex displays an image during the drag, called the *drag proxy*.

When the user moves the drag proxy over another Flex component, that component becomes a possible *drop target*. The drop target inspects the data being dragged, called the *drag source*, to determine whether the data is in a format that the target accepts and, if so, lets the user drop the data onto it. If the drop target determines that the data is not in an acceptable format, the drop target does not permit the drop operation.

Upon a successful drop operation, the data is added to the target and, optionally, can be deleted from its original location.

Drag-and-drop events

Flex uses events to control drag-and-drop operations. Several of these events apply to the drag initiator, and other events apply to the drop target.

The drag initiator events include the following:

- `mouseDown` Broadcast to the drag initiator when the user selects the object with the mouse and holds down the mouse button. You typically use this event to initiate the drag-and-drop operation.
- `dragComplete` Broadcast to the drag initiator when the drag operation completes, either when you drop the drag data onto a drop target or when you end the drag-and-drop operation without performing a drop operation. You can use this event to perform any final cleanup of the drag-and-drop operation. For example, if you drag a List control item from one list to another, you can delete the List control item from the source if you no longer need it.
- `dragBegin` Broadcast to the drag initiator when the user makes a gesture that starts a drag-and-drop operation. No Flex components respond to this event; it is for use by any custom components that you create.

The drop target events include the following:

- `dragEnter` Broadcast to the drop target when a drag initiator passes over the target. Only components that define a handler for this event can be drop targets. Within the handler, you can change the appearance of the drop target to provide visual feedback to the user that the component can accept the drag operation. For example, you can draw a border around the drop target, or give focus to the drop target.
- `dragOver` Broadcast to the drop target when the user moves the mouse over the target. You can handle this event if you want to perform additional logic before allowing the drop operation, such as dropping data to various locations within the drop target, reading keyboard input to determine if the drag-and-drop action is a move or copy of the drag data, or providing different types of visual feedback based on the type of drag-and-drop action.
- `dragDrop` Broadcast to the drop target when the mouse is released over it. You use this event handler to add the drag data to the drop target.
- `dragExit` Broadcast to the drop target when the user drags outside the drop target, but does not drop the data onto the target. You can use this event to restore the drop target to its normal appearance if you modified its appearance as part of handling the `dragEnter` event.

The following steps define the drag-and-drop operation:

1. A component detects that a drag-and-drop operation can be started on it. The component becomes the drag initiator. Typically, you use the `mousedown` event to start the drag-and-drop operation.
 - a Within the handler, the drag initiator creates an instance of the `mx.core.DragSource` class that contains the drag data, and specifies the formats for the data.
 - b Within the handler, call the `mx.managers.DragManager.doDrag()` method, to initiate the drag-and-drop operation.
2. While the mouse button is still down, the user moves the mouse around the application. Flex displays the drag proxy image in your application.

Note: Releasing the mouse button when the drag proxy is not over a target ends the drag-and-drop operation. Flex generates a `DragComplete` event on the drag initiator, and sets the `action` property of the event object to `DragManager.NONE`.

3. When the user moves the drag proxy over a Flex component, Flex broadcasts a `dragEnter` event to the component. If the component does not define a `dragEnter` event handler, it cannot be a drop target.
4. If the component defines a `dragEnter` event handler, the handler examines the `DragSource` object to determine whether the drag data is in an accepted data format. If so, the `dragEnter` event handler sets the `event.handled` property to `true` to signal that it can accept the drop.
5. If the drop target does not accept the drop, the drop target component's parent chain is examined to determine if any component in the chain accepts the drop data.
6. If the drop target accepts the drop data, as determined by setting the `event.handled` property to `true`, Flex broadcasts the `dragOver` event to the target.
7. If the user moves the drag proxy outside of the drop target, Flex broadcasts a `dragExit` event to the drop target. In this case, the user decided not to drop the data onto the drop target.
8. If the user releases the mouse while over the drop target, Flex broadcasts a `dragDrop` event to the drop target. The `dragDrop` event handler adds the drag data to the target.
9. If the user drops the drag data onto a target, Flex broadcasts a `dragComplete` event to the drag initiator.

Initiating a drag-and-drop operation

To initiate a drag-and-drop operation, you call the `DragManager.doDrag()` method from within the event handler for the `dragBegin` or `mousedown` events, as the following example shows. In this example, the drag initiator is a `Canvas` container.

```
<mx:Script>
<![CDATA[
    // Import the DragManager and DragSource classes.
    import mx.managers.DragManager;
    import mx.core.DragSource;

    // Handle the drag initialization for a mousedown event.
    function dragIt(event, text, format) {
```

```

// Create an instance of DragSource.
var ds:DragSource = new DragSource();

ds.addData(text, format);
DragManager.doDrag(event.target, ds, mx.containers.Canvas,
    {backgroundColor:event.target.backgroundColor, width:30, height:30},
    undefined, undefined, 30);
}
]]>
</mx:Script>

```

This event handler takes the following arguments:

- **event** An object that contains a reference to the drag initiator
- **text** An object that represents the data to drag (in this case, a text string)
- **format** A String that contains the format of the data

The **format** argument is a text string that you use to label a data format, such as “list data” or “grid data.” The drop target examines this string to determine whether the data format matches a format that it accepts. If the format matches, you can drop the data on the target; if the format doesn’t match, you cannot.

Within the event handler, you first create an instance of a `DragSource` object, and then initialize it with the drag data and the format.

The event handler calls the `DragManager.doDrag()` method. This method has the following signature:

```
doDrag(dragInitiator, dragSource, dragImage, imageInitObj,
    xOffset, yOffset, imageAlpha)
```

where:

- **dragInitiator** An Object that specifies the component that initiates the drag operation. This argument is required.
- **dragSource** A `DragSource` object that contains the data to drag. This argument is required.
- **dragImage** A String or Function that specifies the drag proxy, which is the image of the drag initiator that you see as you drag it. In the preceding example, you specify the `Canvas` container as the object.

To specify a symbol, use a string that specifies the symbol’s name. To specify a class, pass in the constructor for the class. This argument is optional. If omitted, Flex uses a standard drag rectangle.

For example, to specify a JPEG file named `atom.jpg` as the drag icon, you can use the following statement:

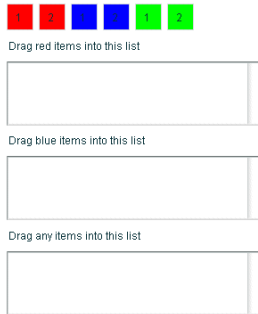
```
DragManager.doDrag(event.target, ds, 'atom.jpg');
```

- **imageInitObj** An Object that specifies the Initialization object sent to `dragImage`. This argument is optional.
- **xOffset** A Number that specifies the *x* offset, in *dragInitiator* coordinates, for `dragImage`. This argument is optional.

- *yOffset* A Number that specifies the *y* offset, in *dragInitiator* coordinates, for *dragImage*. This argument is optional.
- *imageAlpha* A Number that specifies the alpha value used for *dragImage*. This argument is optional. If omitted, Flex uses an alpha value of 50, where a value of 0 corresponds to transparent and a value of 100 corresponds to fully opaque.

Example drag-and-drop operation

The following figure shows an example that lets you drag colored Canvas containers into a List control:



In this example, you can only drag red boxes into the first List control, blue boxes into the second, and any colored boxes into the third. The following MXML code defines the Canvas containers and the three List controls. The ActionScript code for the event handlers is shown later.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" width="400"
    height="600" marginLeft="6" marginRight="6" initialize="appInit()">
```

```
<!-- Script block goes here. -->
```

```
<mx:Tile width="175">
    <mx:Canvas backgroundColor="#FF0000" borderStyle="solid" width="30"
        height="30" mouseDown="dragIt(event, 'Red 1', 'red')">
        <mx:Label text="1" x="8" y="6" width="22" height="24"/>
    </mx:Canvas>
    <mx:Canvas backgroundColor="#FF0000" borderStyle="solid" width="30"
        height="30" mouseDown="dragIt(event, 'Red 2', 'red')">
        <mx:Label text="2" x="8" y="6" width="22" height="24"/>
    </mx:Canvas>
    <mx:Canvas backgroundColor="#0000FF" borderStyle="solid" width="30"
        height="30" mouseDown="dragIt(event, 'Blue 1', 'blue')">
        <mx:Label text="1" x="8" y="6" width="22" height="24"/>
    </mx:Canvas>
    <mx:Canvas backgroundColor="#0000FF" borderStyle="solid" width="30"
        height="30" mouseDown="dragIt(event, 'Blue 2', 'blue')">
        <mx:Label text="2" x="8" y="6" width="22" height="24"/>
    </mx:Canvas>
    <mx:Canvas backgroundColor="#00FF00" borderStyle="solid" width="30"
        height="30" mouseDown="dragIt(event, 'Green 1', 'green')">
```

```

        <mx:Label text="1" x="8" y="6" width="22" height="24"/>
    </mx:Canvas>
    <mx:Canvas backgroundColor="#00FF00" borderStyle="solid" width="30"
        height="30" mouseDown="dragIt(event, 'Green 2', 'green')">
        <mx:Label text="2" x="8" y="6" width="22" height="24"/>
    </mx:Canvas>
</mx:Tile>
<mx:VBox width="100%" height="100%">
    <mx:Label text="Drag red items into this list" />
    <mx:List dragEnter="doDragEnter(event, 'red')"
        dragExit="doDragExit(event);"
        dragOver="doDragOver(event);"
        dragDrop="doDragDrop(event, 'firstList', ['red'])"
        width="100%"
        height="33%"
        id="firstList" />
    <mx:Label text="Drag blue items into this list" />
    <mx:List dragEnter="doDragEnter(event, 'blue')"
        dragExit="doDragExit(event);"
        dragOver="doDragOver(event);"
        dragDrop="doDragDrop(event, 'secondList', ['blue'])"
        width="100%"
        height="33%"
        id="secondList" />
    <mx:Label text="Drag any items into this list" />
    <mx:List dragEnter="doDragEnter(event, 'any')"
        dragExit="doDragExit(event);"
        dragOver="doDragOver(event);"
        dragDrop="doDragDrop(event, 'thirdList', ['red','blue','green'])"
        width="100%"
        height="33%"
        id="thirdList" />
</mx:VBox>
</mx:Application>

```

Each Canvas container defines a mouseDown event handler. This event handler initiates a drag-and-drop operation when the user selects the Canvas container and holds down the mouse button.

Each List control defines handlers for the dragEnter, dragExit, dragOver, and dragDrop events. Within these event handlers, the List control determines whether a drag initiator is in the correct format and, if so, handles the drop operation.

The <mx:Script> block for this example is as follows:

```

<mx:Script>
    <![CDATA[
        // Import the DragManager and DragSource classes.
        import mx.managers.DragManager;
        import mx.core.DragSource;

        // Handle the drag initialization for the Canvas containers
        // in the mouseDown event.
        function dragIt(event, text, format) {
            var ds:DragSource = new DragSource();

```

```

        ds.addData(text, format);
        DragManager.doDrag(event.target, ds, mx.containers.Canvas,
            {backgroundColor:event.target.backgroundColor, width:30, height:30},
            undefined, undefined, 30);
    }

    // Handle the dragEnter event for the List controls.
    function doDragEnter(event, format) {
        if (event.dragSource.hasFormat(format) || format == "any")
        {
            event.handled = true;
            event.target.drawFocus(true);
        }
    }

    // Handle the dragExit event for the List controls.
    function doDragExit(event) {
        event.target.drawFocus(false);
    }

    // Handle the dragOver event for the List controls.
    function doDragOver(event) {
        if (Key.isDown(Key.CONTROL))
            event.action = DragManager.COPY;
        else if (Key.isDown(Key.SHIFT))
            event.action = DragManager.LINK;
        else
            event.action = DragManager.MOVE;
    }

    // Handle the dragDrop event for the List controls.
    function doDragDrop(event, target, formats) {
        var prefix:String = "";

        if (event.action == DragManager.COPY)
            prefix = "Copy of ";
        else if (event.action == DragManager.LINK)
            prefix = "Link to ";

        // Since the drag is over, remove focus from the target.
        doDragExit(event);

        for (var i = 0; i < formats.length; i++)
        {
            var data = event.dragSource.dataForFormat(formats[i]);

            if (data != undefined)
                this[target].addItem(prefix + data);
        }
    }

    // Initialize the List controls.
    function appInit() {
        firstList.dataProvider = [];
    }

```

```

        secondList.dataProvider = [];
        thirdList.dataProvider=[];
    }

]]>
</mx:Script>

```

The following sections describe the `ActionScript` code in more detail.

Handling the `dragEnter` event

The drop target must define a handler for a `dragEnter` event for it to be a target. Within the event handler, you use the format information in the `DragSource` object to determine whether the drag data is in a format accepted by the drop target.

In this example, the event handler takes an argument from the drop target that defines the format of the data that it accepts. It then uses the `DragSource.hasFormat()` method to determine whether the `DragSource` object contains data in the accepted format. If the drop target can accept the drop, the handler sets the `event.handled` property to `true`. Otherwise, you cannot drop the data onto the target. The handler also sets focus on the drop target to provide visual feedback to the user that the `List` control accepts a drop operation.

Handling the `dragOver` event

The handler for the `dragOver` event is optional; you do not need to define it to perform a drag-and-drop operation. One reason to define it is to determine if the drag action is a copy, move, or link action. For a copy action, you drop data onto the target, but leave the initiator unchanged. In a move action, you delete the drag data after dropping it.

In this example, the handler determines whether the user is pressing a key to determine the drag action.

Handling the `dragDrop` event

The drop target defines a handler for the `dragDrop` event to handle the actual drop operation. The way you write this handler depends on the specific type of the target component. In the example, the target is a `List` control and uses the `addItem()` method to add the drag data to the drop target.

Handling the `dragExit` event

The handler for the `dragExit` event performs any cleanup on the drop target if the user decides not to drop the data onto it. In the example, the handler removes focus from the target to signal to the user that the drop operation has completed.

The handler for the `dragDrop` event also calls this handler to remove focus from the drop target in the case where the user does drop the drag data onto it. You can call any user-defined function to remove focus from the target; you are not required to call the `dragExit` event handler just for that purpose.

Handling the dragComplete event

You can optionally define a handler for the `dragComplete` event. This handler can perform any final cleanup on the drag initiator, if necessary. For example, in the case of a move operation, the handler can delete the initiator or data from the initiator since you moved it to the target.

Using a container as a drop target

If you want to use a container as a drop target, you must use the `backgroundColor` property of the container to set a color. Otherwise, the background color of the container is transparent, and the Drag and Drop manager is unable to detect that the mouse pointer is on a possible drop target.

Dragging between SWF files

You can use the Loader and Image controls to load one SWF file from another SWF file, including loading a SWF file that contains a Flex application. If you want to drag items from a drag source in the loading SWF file to a drag target in the loaded SWF file, make sure that your MXML code defines the Loader control or Image control that loads the SWF file after the drag source control in your application. Otherwise, the drag proxy appears behind the drag target when you perform the drag operation.

Alternatively, you can use the `UIObject.swapDepths()` method to switch the depths of the drag source and drag target components to ensure that the drag target is at a lower depth than the drag source.

Specifying the drag proxy

This example uses the `<mx:Image>` tag to load a draggable image into a Canvas container. As you drag the image, the Drag and Drop Manager uses the loaded image as the drag proxy.

To specify the loaded image as the drag proxy, you specify the class name of the drag proxy as `mx:Controls.Image`, and use the `imageInitObj` argument of the `doDrag()` method to specify the source value of the drag proxy.

The `imageInitObj` argument lets you pass initialization properties to the drag proxy. The drag proxy is the `mx:Controls.Image` class, so you can pass to it the same properties that you can set in MXML for the `<mx:Image>` tag, as the following example shows:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:Script>
    <![CDATA[
      import mx.managers.DragManager;
      import mx.core.DragSource;
      var xOff:Number;
      var yOff:Number;

      function dragMe(event, img1, format)
      {
        var ds = new DragSource()
        ds.addData(img1, format);
```

```

        DragManager.doDrag(event.target, ds, mx.controls.Image,
        {source: event.target.source});
    }

    function doDragEnter(event)
    {
        event.handled = true;
        event.action=DragManager.MOVE
    }

    function doDragDrop(event,target1, format)
    {
        myimg.x = target1.mouseX - xOff
        myimg.y = target1.mouseY - yOff
    }

    function myoffset(img){
        xOff = img.mouseX
        yOff = img.mouseY
    }
    ]]>
</mx:Script>

<mx:Canvas id="v1" width="500" height="500" dragEnter="doDragEnter(event)"
dragDrop = "doDragDrop(event,v1, 'img')" borderStyle="solid"
backgroundColor="#DDDDDD">

    <mx:Image id="myimg" source="@Embed('w_p.gif')"
        mouseDown="dragMe(event, 'Image', 'img');myoffset(myimg)"/>
</mx:Canvas>
</mx:Application>

```

Using a List, Tree, or DataGrid control

Flex includes support for the drag-and-drop operation directly in the List, Tree, and DataGrid controls. You can use any of these controls as the drag initiator, which makes the process of dragging and dropping simpler than for other controls.

If the initiator is a List, Tree, or DataGrid control, you set the `dragEnabled` property to `true`; you do not need to define a `dragBegin` or `mouseDown` event. Flex automatically creates a `DragSource` object for the drag operation, and calls the `DragManager.doDrag()` method to initiate the drag.

The `DataSource` object created by Flex contains the following data objects:

- For a List or DataGrid control, the first data object contains a copy of the selected item or items in the List or DataGrid control, and has a format string of `items`. The selected items implement the `DataProvider` API.
- For a Tree control, the first data object contains a copy of the selected item or items in the Tree control, and has a format string of `treeItems`. The selected items implement the `TreeDataProvider` API.
- The second data object contains a copy of the initiator, and has a format string of `source`.

Dragging and dropping using a Tree control

In the following example, you drag items from the Tree control and drop them in a List control:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    width="400" height="220" marginLeft="6" marginRight="6" >

    <mx:Script>
        <![CDATA[
            import mx.managers.DragManager;

            function doDragEnter(event) {
                event.handled = true;
            }

            function doDragExit(event) {
                event.target.hideDropFeedback();
            }

            function doDragOver(event) {
                event.target.showDropFeedback();
                if (Key.isDown(Key.CONTROL))
                    event.action = DragManager.COPY;
                else if (Key.isDown(Key.SHIFT))
                    event.action = DragManager.LINK;
                else
                    event.action = DragManager.MOVE;
            }

            function doDragDrop(event) {
                // Since the drag is over, remove visual feedback from the target.
                doDragExit(event);

                var dragItems = event.dragSource.dataForFormat("treeItems");
                var dest = event.target;
                var dropLoc = dest.getDropLocation();

                dest.clearSelected();
                for (var i = dragItems.length - 1; i >= 0; i--)
                {
                    dest.addItemAt(dropLoc, dragItems[i]);
                    dest.selectItem(dropLoc, true);
                }
            }

            function initApp() {

                firstList.dataProvider = treeDP;
                secondList.dataProvider = [];
            }
        ]]>
    </mx:Script>

    <mx:XML id="treeDP">
```

```

<node label="Mail">
  <node label="Inbox"/>
  <node label="Personal Folder">
    <node label="Demo" isBranch="true" />
    <node label="Personal" isBranch="true" />
    <node label="Saved Mail" isBranch="true" />
    <node label="bar" isBranch="true" />
  </node>
  <node label="Sent" isBranch="true" />
  <node label="Trash"/>
</node>
<node label="Calendar"/>
</mx:XML>

<mx:Label text="Drag items from one list to another" />
<mx:HBox marginBottom="6">
  <mx:Tree id="firstList"
    dragEnabled="true"
    multipleSelection="true"
    initialize="initApp()"/>
  <mx:List
    dragEnter="doDragEnter(event)"
    dragExit="doDragExit(event);"
    dragOver="doDragOver(event);"
    dragDrop="doDragDrop(event)"
    id="secondList" />
</mx:HBox>
</mx:Application>

```

In this example, you populate a `Tree` control from a data model, and set the `dragEnabled` property in the `Tree` control to `true`. You do not have to define any additional events on the `Tree` control to support drag-and-drop operations.

The target of the drag-and-drop operation is the `List` control, which defines event handlers for the `dragEnter`, `dragExit`, `dragOver`, and `dragDrop` events, similar to the previous example.

The handler for the `dragDrop` event does most of the work in this example. Remember that Flex automatically creates a `DataSource` object for the `Tree` control. In the `DataSource` object, the selected items in the `Tree` control are copied into the `DataSource` object with a format string of `items`. Therefore, the following statement writes the selected items to the variable named `dragItems`:

```
var dragItems = event.dragSource.dataForFormat("items");
```

The `dragItems` variable always contains an `Array`. If you drag a single item from the `Tree` control, the array is one item long. If you select multiple `Tree` items, it contains one array entry per item. You use the `addItemAt()` method of the `Tree` control to add the dragged items to the `Tree` control.

Removing a drag item from a List, Tree, or DataGrid control

The item that you drag from a `List`, `Tree`, or `DataGrid` control is actually a copy of the item, not the item itself. Therefore, when you drop the item onto the drop target, the item appears in both the drag initiator and the drop target.

If you want to modify the drag initiator to delete the item, you can add the logic to the event handler for the `dragComplete` event. The following example removes one or more items from a Tree control when the Tree control is the drag initiator:

```
<mx:Script>
  <![CDATA[

    function doDragComplete(event)
    {
      var dragItems = event.dragSource.dataForFormat("source").selectedItems;
      var counter = dragItems.length;
      for (var i=0;i<counter;i++)
      {
        var item = dragItems[i];
        item.removeTreeNode();
      }
    }
  ]]>
</mx:Script>
```

For an example using the DataGrid control, see the Explorer sample application in the `samples.war` file.

Drag and Drop Manager syntax

The following sections describe the syntax of the classes that make up the Drag and Drop Manager.

Class `mx.managers.DragManager` syntax

The following table describes the syntax of the `mx.managers.DragManager` class:

| Property | Type | Use | Description |
|-------------------------|---------|-----------------|---|
| <code>isDragging</code> | Boolean | Static property | Read-only property that returns <code>true</code> if a drag is in progress. |
| <code>NONE</code> | String | Static property | Read-only constant that specifies a drag action. The default value is <code>none</code> . |
| <code>MOVE</code> | String | Static property | Read-only constant that specifies a drag action. The default value is <code>move</code> . |
| <code>COPY</code> | String | Static property | Read-only constant that specifies a drag action. The default value is <code>copy</code> . |
| <code>LINK</code> | String | Static property | Read-only constant that specifies a drag action. The default value is <code>link</code> . |
| <code>doDrag()</code> | | Static method | Initiates a drag-and-drop operation. |

Class `mx.core.DragSource` syntax

The following table describes the syntax of the `mx.core.DragSource` class:

| Property | Type | Use | Description |
|------------------------------|-----------------|----------|--|
| <code>formats</code> | String Array | Property | Read-only property that contains the formats of the drag data. Set this property using the <code>addData()</code> or <code>addHandler()</code> method. The default value depends on the data added to the <code>DragSource</code> object. |
| <code>addData()</code> | | Method | Adds data and a corresponding format string to the drag source. This method does not return a value. The signature of this method is as follows: <code>addData(data, format)</code> <ul style="list-style-type: none">• <i>data</i> (Required) An Object that specifies the drag data. This can be any object, a String, <code>DataProvider</code>, and so on.• <i>format</i> (Required) A String that specifies a label that describes the format for this data. |
| <code>addHandler()</code> | | Method | Adds a handler that is called when data for the specified format is requested. This is useful when dragging large amounts of data. The handler is only called if the data is requested. This method does not return a value. The signature of this method is as follows: <code>addHandler(obj, handler, format)</code> <ul style="list-style-type: none">• <i>obj</i> (Required) An Object that contains the handler.• <i>handler</i> (Required) A function that specifies the handler called to request the data. This function must return the data in the specified format.• <i>format</i> (Required) A String that specifies the format for this data. |
| <code>dataForFormat()</code> | | Method | Retrieves the data for the specified format. If the <code>addData()</code> method added the data, it is returned directly. If the <code>addHandler()</code> method added the data, it calls the handler function to return the data. This method returns an array of Objects containing the data, in the requested format. If you drag a single item, the array is one item long. The signature of this method is as follows: <code>dataForFormat(format)</code> <ul style="list-style-type: none">• <i>format</i> (Required) A String that specifies a label that describes the format for the data to return. |
| <code>hasFormat()</code> | | Method | This method returns <code>true</code> if the data source contains the requested format, and <code>false</code> otherwise. The signature of this method is as follows: <code>hasFormat(format)</code> <ul style="list-style-type: none">• <i>format</i> (Required) A String that specifies a label that describes the format for the data. |

UIObject events

The Drag and Drop Manager adds several new events to the UIObject class. The following table describes these events. For information on the `mouseDown` event, see [Chapter 12, “Working with ActionScript in Flex,”](#) on page 319.

| Event | Description |
|---------------------------|--|
| <code>dragBegin</code> | Broadcast to the drag initiator when the user makes a gesture that starts a drag-and-drop operation. No Flex component generates this event; it is used by any custom components that you create. The handler for this event calls the <code>DragManager.doDrag()</code> method if a drag-and-drop operation occurs. For more information, see “Class <code>mx.managers.DragManager</code> syntax” on page 565. |
| <code>dragComplete</code> | <ul style="list-style-type: none">• Broadcast to the drag initiator component when the drag has completed. If you release the mouse button when the drag proxy is not over a target, you end the drag-and-drop operation. Flex generates a <code>DragComplete</code> event on the drag initiator, and sets the <code>action</code> property of the event object to <code>DragManager.NONE</code>. |
| <code>dragDrop</code> | <ul style="list-style-type: none">• Broadcast to the drop target when the mouse is released over a drop target. |
| <code>dragEnter</code> | <ul style="list-style-type: none">• Broadcast to the drop target when a drag operation passes over the target. If the handler sets <code>event.handled</code> to <code>true</code>, the component becomes a drop target. |
| <code>dragExit</code> | <ul style="list-style-type: none">• Broadcast to the drop target when the user drags outside the target, which means that the drag data will not be dropped on the target. |
| <code>dragOver</code> | <ul style="list-style-type: none">• Broadcast to the drop target when the user moves the mouse over the target and after the <code>dragEnter</code> event handler sets <code>event.handled</code> to <code>true</code>. |

List, DataGrid, and Tree control syntax

The Drag and Drop Manager adds the following optional properties to the List, DataGrid, and Tree controls:

| Property | Type | Use | Description |
|-------------------------------|----------|----------|--|
| <code>dragEnabled</code> | Boolean | Property | Specifies that the control is a drag initiator, <code>true</code> , or not, <code>false</code> (default). When <code>true</code> , you can drag selected items in the control; Flex automatically creates a <code>DragSource</code> object, where: <ul style="list-style-type: none">• The first data object contains a copy of the selected item, or items, in the List, Tree, or DataGrid control, and has a format string of <code>items</code>.• The second data object contains a copy of the initiator, and has a format string of <code>source</code>. |
| <code>dragImage</code> | Function | Property | Read-only property that returns a class name that you can use as a drag image. The default value is <code>mx.controls.listclasses.DragProxy</code> . |
| <code>dragImageInitObj</code> | Object | Property | Read-only property that contains an initialization object that you can use as the <code>imageInitObj</code> argument to the <code>DragManager.doDrag()</code> method. The default value depends on the data being dragged. |

| Property | Type | Use | Description |
|---------------------------------|--------|----------|---|
| <code>dragOffset</code> | Object | Property | Read-only property that contains the offset that you can use as the offset argument to the <code>DragManager.doDrag()</code> method. The default value depends on where the mouse was clicked. |
| <code>dropIndicatorSkin</code> | String | Skin | Specifies the name of the skin element to use for the drop insert indicator. The default value is <code>ListDropIndicator</code> . |
| <code>getDropLocation()</code> | | Method | Returns the item index in the initiator control at the current mouse location. |
| <code>hideDropFeedback()</code> | | Method | Hides drop target feedback and removes the focus rectangle. You typically call this method from within the handler for the <code>dragExit</code> and <code>dragDrop</code> events. This method returns nothing. |
| <code>showDropFeedback()</code> | | Method | Specifies to display the focus rectangle around the target control and positions the drop indicator where the drop operation should occur. If the List, Tree, or DataGrid control has active scrollbars, hovering the mouse pointer over the top or bottom of it scrolls the contents. You typically call this method from within the handler for the <code>dragOver</code> event. This method returns nothing. |

CHAPTER 23

Using the History Manager

The Macromedia Flex History Manager lets users navigate through a Flex application using the web browser's back and forward navigation commands.

Contents

| | |
|--|-----|
| About history management | 569 |
| Using standard history management | 569 |
| Using custom history management | 571 |
| How the HistoryManager class saves and loads state | 574 |
| Using history management in a custom HTML file | 575 |

About history management

The Flex History Manager lets users navigate through a Flex application using the web browser's back and forward navigation commands. For example, a user can navigate through several Accordion container panes in a Flex application, and then click the browser's back button to return the application to its previous states.

By default, Flex enables history management for navigator containers, without using any ActionScript or MXML tags. You can also use the HistoryManager class in ActionScript to provide custom history management for other objects in an application, and to call the HistoryManager class's methods.

Note: History management is not supported on Netscape 4.x and Opera 6.0 web browsers.

Using standard history management

History management is available by default for the Accordion and TabNavigator navigator containers. It is disabled by default for the ViewStack navigator container. When history management is enabled, as the user navigates within different navigator containers within an application, each navigation state is saved. Selecting the web browser's back or forward browser command displays the previous or next navigation state that was saved. History management keeps track of where you are in an application, but it is not an undo and redo feature that remembers what you have done.

Note: When history management is enabled for a particular component, such as a navigator container, only the state of the navigator container is saved. The state of any of the navigator container's child components is not saved unless history management is specifically enabled for that component.

For information about how the navigation state is saved and restored, see [“How the HistoryManager class saves and loads state” on page 574](#).

Flex automatically enables history management for the following navigator containers:

- Accordion
- TabNavigator

To enable history management for a navigator container, you set the container's `historyManagement` property to `true`, as the following example shows:

```
<mx:TabNavigator historyManagement="true">
```

You can disable or enable history management for a navigator container by setting the container's `historyManagement` property to `false` or `true`, respectively. The following example shows a `TabNavigator` container with history management enabled:

```
<mx:TabNavigator historyManagement="true">
```

In the following example, the user's panel sections are saved for the first `Accordion` container because it uses default settings, but the second `Accordion` container has the `historyManagement` property explicitly set to `false`. When the user selects the web browser's back or forward command, the previous or next state is displayed for the first container, but not for the second.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" width="600"
  height="800">

  <!-- History management is enabled by default for this Accordion. -->
  <mx:Accordion width="100%" height="50%">
    <mx:VBox label="View 1">
      <mx:TextInput text="View 1" />
    </mx:VBox>
    <mx:VBox label="View 2">
      <mx:TextInput text="View 2" />
    </mx:VBox>
  </mx:Accordion>

  <!-- History management is disabled for this Accordion. -->
  <mx:Accordion historyManagement="false" width="100%" height="50%">
    <mx:VBox label="View 1">
      <mx:TextInput text="View 1" />
    </mx:VBox>
    <mx:VBox label="View 2">
      <mx:TextInput text="View 2" />
    </mx:VBox>
  </mx:Accordion>
</mx:Application>
```

You can disable history management for an entire application by requesting a *myapp.mxml.swf* file directly or in a custom HTML wrapper page that does not use history management. For more information about customizing the HTML wrapper, see [Chapter 38, “Deploying Applications,” on page 835](#).

Using custom history management

You can register a component with the `HistoryManager` class if it implements the `mx.managers.StateInterface` interface. All of the navigator containers implement the `StateInterface` interface in their class definitions. The `StateInterface` interface contains two methods: `saveState()` and `loadState()`. As their names imply, these methods save and load a component's navigation states.

The `HistoryManager` class contains a `load()` method that calls the `loadState()` method for each registered component with an object identical to the one that the `saveState()` method returns.

Registering a component with the `HistoryManager` class

To register a component with the `HistoryManager` class, you call the `HistoryManager` class's `register()` method with a reference to a component instance that implements the `StateInterface` interface. Each registered component receives a unique state ID based on its full pathname. In the following example, the `Application` component (`this`) is registered with the `HistoryManager` class when the `Application` is initialized:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    initialize="mx.managers.HistoryManager.register(this);">
```

Implementing the `saveState()` and `loadState()` methods

To use the `HistoryManager` class for a registered component, you must implement the `StateInterface` `saveState()` and `loadState()` methods to save and load the state information you want. The `saveState()` method returns an object that contains property:value pairs that represent the current navigation state of a component. An application's total navigation state is limited to the maximum URL size supported by the user's web browser, so you should write the `saveState()` method for a component to save the least amount of data possible. For example, you can write a `saveState()` method for a `List` control that saves just the `selectedIndex` property.

The following example contains `saveState()` and `loadState()` methods for an `Application` component. The `saveState()` method saves the ZIP code used to call a web service that provides stock market information. When the user selects the web browser's back or forward command, the `loadState()` method compares the current ZIP code value to the saved ZIP code value. If the saved value is different, the `loadState()` method calls the web service using the saved ZIP code value.

```
<mx:Application implements="mx.managers.StateInterface"
    initialize="initBrowser()">
    <mx:Script>
        <![CDATA[
            ...
```

```

import mx.managers.HistoryManager;

function initBrowser() {
    // Register with the HistoryManager.
    HistoryManager.register(this);
}

public function saveState():Object
{
    var state = new Object();
    if (myZip.text != undefined) {
        state.zipCode = myZip.text;
    }
    if (myTicker.text != undefined) {
        state.ticker = myTicker.text;
    }
    return state;
}

public function loadState(state:Object)
{
    if (state.zipCode == undefined) {
        resetGetStockQuotes();
    } else if (state.zipCode != myZip.text) {
        myZip.text = state.zipCode;
        ws.GetStockQuotes.send();
    }
    if (state.ticker == undefined) {
        resetStocks();
    } else if (state.ticker != myTicker.text) {
        myTicker.text = state.ticker;
        ws2.GetStockQuotes.send();
    }
}
]]>
</mx:Script>
...
</mx:Application>

```

The following example is an MXML component, `Browser.mxml`, that registers with the `HistoryManager` and implements the `saveState()` and `loadState()` methods. The component lets the user browse through a set of images.

```

<?xml version="1.0"?>
<mx:HBox xmlns:mx="http://www.macromedia.com/2003/mxml"
    verticalAlign="middle" width="300" height="150" initialize="initBrowser()">
    <mx:Script>
        <![CDATA[
            import mx.managers.HistoryManager;

            var data;

            function initBrowser() {

```

```

        // Register with the HistoryManager.
        HistoryManager.register(this);

        // Select the first image by default.
        imageList.selectedIndex = 0;
        selectImage(false);
    }

    function selectImage(bSaveState:Boolean) {
        holder.contentPath =
            imageList.dataProvider[imageList.selectedIndex].image;

        if (bSaveState)
            HistoryManager.save();
    }

    function saveState():Object {
        var state = new Object();

        state.selectedIndex = imageList.selectedIndex;

        return state;
    }

    function loadState(state:Object) {
        var newIndex = state.selectedIndex;

        if (newIndex == undefined)
            newIndex = 0;

        if (newIndex != imageList.selectedIndex) {
            imageList.selectedIndex = newIndex;
            selectImage(false);
        }
    }
}]]>
</mx:Script>
<mx:List id="imageList" dataProvider="{data}"
    width="150" height="130" change="selectImage(true)" />
<mx:Spacer width="20" />
<mx:Loader id="holder" width="50" height="50"/>
</mx:HBox>

```

The following example shows an application file that uses the Browser component. The Array of data defined in the `<mx:Array id="data1">` element is bound to the Browser component's data property.

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" xmlns:local=""
    width="400" height="400" >

    <mx:Array id="data1">
        <mx:Object>
            <label>Product1</label>
            <image>images/digital1.jpg</image>

```

```

        </mx:Object>
        <mx:Object>
            <label>Product2</label>
            <image>images/digital2.jpg</image>
        </mx:Object>
        <mx:Object>
            <label>Product3</label>
            <image>images/digital3.jpg</image>
        </mx:Object>
    </mx:Array>

    <local:Browser data="{data1}" />
</mx:Application>

```

Calling the HistoryManager class's static methods

When you register a component with the HistoryManager class, the HistoryManager `save()` method is invoked automatically when the user navigates through an application and uses the web browser's back and forward navigation commands.

The `save()` method and the `register()` and `unregister()` methods, which let you register and unregister a component, are static methods that you can call from your ActionScript code. The following table describes these methods:

| Method | Description |
|---|--|
| <code>register(StateInterface)</code> | Registers a component with the HistoryManager class; for example: <code>HistoryManager.register(myList);</code> |
| <code>save()</code> | Saves the current navigation state of all components registered with the HistoryManager class; for example: <code>HistoryManager.save();</code> |
| <code>unregister(StateInterface)</code> | Unregisters a component from the HistoryManager class; for example: <code>HistoryManager.unregister(myList);</code> |

How the HistoryManager class saves and loads state

The history management feature uses the Macromedia Flash `getURL()` function to load an invisible HTML frame in the current web browser window. It then encodes a Flex application's navigation states into the invisible frame's URL query parameters. A SWF file, called `history.swf`, in the invisible frame decodes the query parameters and sends the navigation state back to the HistoryManager class. This section describes how the HistoryManager class encodes navigation data into a URL query string and then decodes that navigation data to restore navigation states.

Encoding navigation state data

The `HistoryManager` class's `save()` method collects the state object returned by the `StateInterface.saveState()` method for each registered component. The `save()` method encodes each property of each object into a query string that uses the standard `prop1=value1&prop2=value2` format. The state ID of the appropriate registered component is added to each property name to identify which component each property belongs to. For example, for a `TabNavigator` container with a state ID of 4 that returns the following state object:

```
{selectedIndex:5}
```

The query string is:

```
id4_selectedIndex=5
```

Decoding and restoring navigation state data

The `history.swf` file passes stored state properties to a Flex application in a single object. The `HistoryManager` class extracts the state IDs from the properties in this object and rebuilds state objects for each registered component. The state objects are constructed from the `Application` object down through its children. For example, when an `Accordion` container is the third item in a `ViewStack` container, the `ViewStack` container must be set to its third item before the `Accordion` container's navigation state is restored.

Using history management in a custom HTML file

When you place a Flex application inside a custom HTML page instead of generating the HTML page automatically, you must set up the HTML page to support history management if you want to use it.

The following steps are required to support history management:

1. Include the following text at the top of the HTML document:

```
<script language='javascript' charset='utf-8'  
  src='/flex/flex-internal?action=js'></script>
```

2. Add the `historyUrl` and `lconid` parameters to the `flashVars` variable for both the object and embed tags, as the following example shows. You must add these parameters in JavaScript, because history management uses a JavaScript variable called `lc_id`.

```
document.write(" <param name='flashVars' value='historyUrl=%2Fflex%2Fflex-  
  internal%3Faction%3Dhtml&lconid=" + lc_id + "'>");
```

3. Add the `_history` iframe, as the following example shows:

```
<iframe src='/flex/flex-internal?action=html' name='_history'  
  frameborder='0' scrolling='no' width='22' height='0'></iframe>
```


CHAPTER 24

Improving Startup Performance

Macromedia Flex provides settings that let you determine when controls and other components are created when you invoke a Flex application. You can use these settings to reduce the startup display of your applications, or stagger the display so that parts of the application appear before the entire application loads.

Flex also lets you hide the children of a Panel container when resizing the container. This lets you avoid the jerky animation that can occur when Flex cannot update the screen quickly enough.

Contents

| | |
|---|-----|
| About component instantiation | 577 |
| Using the creationPolicy property | 578 |
| Manually instantiating controls | 582 |
| Using the childDescriptors property | 584 |
| Setting container creation order | 587 |

About component instantiation

By default, containers create only the controls that initially appear to the user. Flex creates the other controls, or *descendants*, later, if the user navigates to them. Containers with a single view, such as Box, Form, and Grid containers, create all of their descendants because these containers display all of their descendants immediately.

Containers with multiple views, called *navigator containers*, only display the descendants that are visible at any given time. When navigator containers such as the ViewStack and Accordion containers are created, they do not immediately create all of their descendants, but only those descendants that are initially visible. Flex defers the instantiation of descendants that are not initially visible until the user navigates to a view that contains them.

The result of this deferred instantiation of navigator containers is that an MXML application with navigator containers loads quickly, but the user experiences brief pauses when he or she moves from one view to another.

You instruct your application to instantiate every control at application startup using a single MXML property. This lets you override the default way that Flex handles instantiation. You can also completely control the instantiation process by using an ActionScript Application Programming Interface (API) for instantiation. This API lets you define the order of instantiation or programmatically instantiate controls at any time.

Using the `creationPolicy` property

Every container has a `creationPolicy` property that determines how the container decides which of its descendants, if any, to create when the container is created. You can change the policy of a container using MXML or ActionScript.

Classes that descend from the `UIObject` class that do not specify a `creationPolicy` property inherit their parent's `creationPolicy` property.

The possible values for the `creationPolicy` property are `auto`, `all`, `none`, and `queued`. The meaning of these settings depends on whether the container is a navigator container (multiple-view container) or a single-view container.

The `creationPolicy` property is not inheritable. This means that if you set the value of the `creationPolicy` property to `none` on an outer container, all containers within that container will have the default value of the `creationPolicy` property unless otherwise set. In addition, if you have two containers at the same level (of the same type) and you set the `creationPolicy` of one of them, the other container will have the default value of the `creationPolicy` property unless you explicitly set it.

Single-view containers

The following table describes the values of the `creationPolicy` property when used with single-view containers:

| Value | Description |
|------------------------|--|
| <code>all, auto</code> | Creates all controls in the single-view container. The default value is <code>auto</code> , but <code>all</code> results in the same behavior. |
| <code>none</code> | <p>Instructs Flex to not instantiate any component within the container until instantiation methods are explicitly called.</p> <p>When the value of the <code>creationPolicy</code> property is <code>none</code>, you should explicitly set a width and height for that container so that the application determines the appropriate size of that container. Normally, the container is scaled to fit the children that are inside it, but since no children are created, proper scaling is not possible. If you do not explicitly resize the container, it grows to accommodate the children when they are created.</p> <p>For more information on manually instantiating controls, see “Manually instantiating controls” on page 582.</p> |
| <code>queued</code> | <p>Adds the container to a queue of other containers that have the value of the <code>creationPolicy</code> property of <code>queued</code>. After all containers are created, Flex creates the children of the <code>queued</code> containers in the order in which they appear in the application, unless you specify a <code>creationIndex</code> property. For more information on using the <code>queued</code> <code>creationPolicy</code> property, see “Setting container creation order” on page 587.</p> |

The following example sets the value of a VBox container's `creationPolicy` property to `all`:

```
<?xml version="1.0"?>
<mx:Application width='500' height='400'
  xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:VBox id="myVBox" height="100" width="125" creationPolicy="all">
    <mx:Button id="mybutton" label="Get Weather" click="ws.getTemp.send();" />
  </mx:VBox>
</mx:Application>
```

This example does not require a `creationPolicy` property because it is a single-view container and the default behavior is to create all children. However, it is provided to show you the syntax.

The default behavior of all single-view containers is that they and their children are entirely instantiated when the application starts. If you set the `creationPolicy` property to `none`, however, you can then selectively instantiate controls within the containers. For more information, see [“Manually instantiating controls” on page 582](#).

Multiple-view containers

The following containers have multiple views and, so, are defined as navigator containers:

- `ViewStack`
- `TabNavigator`
- `Accordion`

When you instantiate a multiple-view container, Flex creates all of the top-level children. For example, creating an `Accordion` container triggers the creation of every pane. The `creationPolicy` property determines the creation of the child controls inside each panel.

When you set the `creationPolicy` property to `auto` (the default value), navigator containers instantiate only the controls and their children that appear in the initial view. The first panel of the `Accordion` container is the initial panel view, as the following figure shows:

The image shows a screenshot of a web application interface. At the top, there is a green header bar with the text "1. Shipping Address". Below this header, there are several input fields: "First Name", "Last Name", "Address" (with two stacked text boxes), "City", "Phone", "State" (a dropdown menu currently showing "AK"), and "Zip Code". Below the "Zip Code" field is a "Continue" button. At the bottom of the form, there are three more header bars: "2. Billing Address", "3. Credit Card Information", and "4. Submit Order", all in a light gray color.

When the user navigates to another panel in the Accordion container, the navigator container creates the next set of controls, and recursively creates the new view's controls and their descendants.

If you set the Accordion container's `creationPolicy` property to `all`, the navigator container creates all controls and their descendants for all panels in the Accordion container when the application starts. This results in longer startup time for the application, but quicker response time for user navigation.

The following table describes the values of the `creationPolicy` property when you use it with navigator containers:

| Value | Description |
|---------------------|--|
| <code>all</code> | Creates all controls in all views of the navigator container. This setting causes a delay in application startup time, but results in quicker response time for user navigation. |
| <code>auto</code> | Creates all controls only in the initial view of the navigator container. This setting causes a faster startup time for the application, but results in slower response time for user navigation. This setting is the default for multiple-view containers. |
| <code>none</code> | Instructs Flex to not instantiate any component within the navigator container or any of the navigator container's panels until instantiation methods are explicitly called. For more information on manually instantiating components, see “Manually instantiating controls” on page 582 . |
| <code>queued</code> | Has no effect on navigator containers. However, you can use the <code>queued creationPolicy</code> property on child containers of the navigator container. In these cases, the behavior of the container is the same as a single-view container. For more information on the <code>queued creationPolicy</code> for single-view containers, see “Single-view containers” on page 578 . |

The following example sets the `creationPolicy` property of an Accordion container to `all`, which instructs the container to instantiate all controls for every panel in the navigator container when the application starts:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:Accordion id="myAccordion" creationPolicy="all" >
    ...
  </mx:Accordion>
  ...
</mx:Application>
```

Initializing controls in navigator containers

The `initialize` event handlers of containers have an impact on navigator containers. In order to understand the impact, you must understand the order in which components are initialized in Flex.

In the following example, Flex defers the instantiation of the `button1` control until the user navigates to the `vbox1` VBox container:

```
<mx:Application>
  <mx:ViewStack id="viewStack0">
```

```

    <mx:VBox id="vbox0">
        <mx:Button id="button0"/>
    </mx:VBox>
    <mx:VBox id="vbox1">
        <mx:Button id="button1"/>
    </mx:VBox>
</mx:ViewStack>
</mx:Application>

```

On startup, Flex calls the `initialize` events in the following order:

1. `vbox0`
2. `vbox1`
3. `viewStack0`
4. `Application`
5. `button0`

In addition to the initial view, Flex creates the top-level navigator container, but defers the child views and the controls inside those child views until the user navigates to that view.

Flex does, however, call the `initialize` event on each view when it first creates the navigator container. Code inside the event handler that references the view's children does not work, because the children haven't been instantiated yet. To avoid this situation, you can use the `childrenCreated` event instead of the `initialize` event to perform initialization functions. Flex triggers the `childrenCreated` event when the view's children are instantiated.

The following example defines the `childrenCreated` event in an `Accordion` container:

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">

    <mx:Accordion id="acc1" creationPolicy="none">
        <mx:Canvas id="c0" childrenCreated="childrenCreatedNotification()">
            <mx:Label id="l1" text=""/>
            <mx:Button id="b1" label="My Button"/>
        </mx:Canvas>
    </mx:Accordion>

    <mx:Script>
        <![CDATA[
            function createComp() {
                acc1.createComponent(0);
            }

            function childrenCreatedNotification() {
                tal.text = "Children created";
            }
        ]]>
    </mx:Script>

    <mx:TextArea id="tal" />
    <mx:Button label="My Other Button" click="createComp()" />
</mx:Application>

```

Uninstantiating objects

After you instantiate a component, it continues to exist until the user quits the application or you call the `mx.core.View` class's `destroyChild()`, `destroyChildAt()`, or `destroyAllChildren()` methods. For more information on using these methods, see the `View` class in *Flex ActionScript and MXML API Reference*.

Manually instantiating controls

Flex provides an API that lets you manually instantiate the controls in a container. You can use these methods to control the instantiation of controls in single-view containers and navigator containers.

To manually control instantiation, you set the `creationPolicy` property of the container to `none`, and then call the `createComponent()` or `createComponents()` methods to instantiate controls in Flex applications. The following sections describe these methods.

Note: Complete instantiation of some `UIObject` class objects can take more than a single draw and refresh of Macromedia Flash Player. This might result in the methods returning before the child is in its final state.

Using the `createComponent()` method

The `createComponent()` method instantiates the specified child of the specified container. This method does not recursively instantiate any children of the specified child by default. If the child has already been instantiated, this method does not instantiate it again, but instead returns the already instantiated child.

The `createComponent()` method has the following signature:

```
container.createComponent(index|childDescriptor [, recursionFlag]):Object
```

The `index` argument is a number that specifies one of the child descriptors in a given container. For example, if an `HBox` container contains five text fields, each with an `id` property, you can refer to the first as 0, the second as 1, and so on. Controls do not have to be instantiated to have an index entry.

The `childDescriptor` argument is similar to `index`, but provides additional information about the child control. For more information on using `childDescriptor`, see [“Using the childDescriptors property” on page 584](#).

The optional argument `recursionFlag` determines whether Flex should instantiate children of the specified component. Set the parameter to `true` to instantiate children of the specified component, or `false` to not instantiate the children.

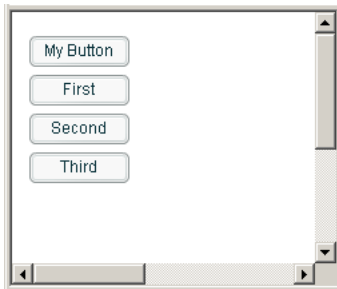
The `createComponent()` method creates a child at the end of the container's list of children. You can use the `setChildIndex()` method to change the index, if necessary.

The following example sets the `creationPolicy` property in a VBox container to `none`. Each time the user clicks a new button, Flex instantiates the next child in the VBox container using the `createComponent()` method. However, because the recursion flag is set to `false`, Flex does not instantiate any children of the VBox container's child controls (in this example there are none):

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
  <mx:Button id="myButton" label="My Button" width="75" click="inst(0);" />
  <mx:VBox id="vb" creationPolicy="none">
    <mx:Button id="b0" label="First" width="75" x="0" click="inst(1);" />
    <mx:Button id="b1" label="Second" width="75" y="50" click="inst(2);" />
    <mx:Button id="b2" label="Third" width="75" y="100" />
  </mx:VBox>

  <mx:Script>
    <![CDATA[
      function inst(n) {
        vb.createComponent(n, false);
      }
    ]]>
  </mx:Script>
</mx:Application>
```

After the application runs to completion, Flex displays the initial button (labeled My Button) and the three child buttons of the VBox container, as the following figure shows:



Using the `createComponents()` method

The `createComponents()` method instantiates some or all descendants of the specified container, according to the container's `creationPolicy` property.

The `createComponents()` method has the following signature:

```
container.createComponents():Void
```

On a single-view container, calling the `createComponents()` method instantiates all controls in that container, regardless of the value of the `creationPolicy` property.

In navigator containers, if you set the `creationPolicy` property to `all`, calling the `createComponents()` method creates all controls in all views of the container. If you set the `creationPolicy` property to `none` or `auto`, calling the `createComponents()` method creates only the current view's controls and their descendents.

Note: Initially, you must set the `creationPolicy` property to `none` or `auto`, otherwise all components are created anyway, and the `createComponents()` method is not necessary. You can change the value of the `creationPolicy` property of any container using `ActionScript`.

The following example does not instantiate any of the buttons in an `HBox` container at startup, but does so only when the user changes the `creationPolicy`. The user does this by selecting *all* from the drop-down list and clicking the Change Policy button. The script block shows that you use the `createComponents()` method to instantiate objects based on the current policy of the container.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:ComboBox id="policy">
    <mx:dataProvider>
      <mx:Array>
        <mx:String>none</mx:String>
        <mx:String>all</mx:String>
      </mx:Array>
    </mx:dataProvider>
  </mx:ComboBox>

  <mx:Button label="Change Policy" click="changePolicy();" />
  <mx:HBox id="hb" creationPolicy="none">
    <mx:Button label="1" width="50" y="0" x="0" />
    <mx:Button label="2" width="50" y="0" x="75" />
    <mx:Button label="3" width="50" y="0" x="150" />
  </mx:HBox>

  <mx:Script>
    <![CDATA[
      function changePolicy() {
        var polType = policy.value;
        hb.creationPolicy = polType;
        if (polType == "none") {
        } else if (polType == "all") {
          hb.createComponents();
        }
      }
    ]]>
  </mx:Script>
</mx:Application>
```

Using the `childDescriptors` property

When a Flex application starts, initially there are no controls. Instead, every container has an `Array` of objects that each contain the MXML information of the child controls in the container. Depending on the value of the `creationPolicy` property, Flex immediately begins instantiating controls or it defers the instantiation. If instantiation is deferred, you can use the properties of this `Array` to control the instantiation process.

Each object in the `Array` is a *child descriptor*. You can access this `Array` using a container's `childDescriptors` property, and use a zero-indexed value to identify which descriptor you want.

The following example accesses the `type` property of the `myTile` container's first `childDescriptors` property:

```
var t = myTile.childDescriptors[0].type;
```

The following table describes the public properties of the `childDescriptors` property:

| Property | Description |
|-------------------------|--|
| <code>id</code> | The MXML <code>id</code> of the child control, if an <code>id</code> property was assigned in the MXML tag; otherwise, the ActionScript compiler generates an <code>id</code> . For more information, see “Using the <code>childDescriptors.id</code> property” on page 585 . |
| <code>properties</code> | A plain object that stores the MXML-specified properties and children of the child control; otherwise, the ActionScript compiler generates an <code>id</code> . For more information, see “Using the <code>childDescriptors.properties</code> property” on page 586 . |
| <code>type</code> | A reference to the constructor function for the child control. For more information, see “Using the <code>childDescriptors.type</code> property” on page 586 . |

The `childDescriptors` property points to an Array of objects, so you can use Array functions, such as `length`, to iterate over the children, as the following example shows:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

    <mx:Script>
        <![CDATA[
            function iterateOverChildren() {
                var n = tile.childDescriptors.length;
                for (var i = 0; i < n; i++) {
                    // Process the child controls.
                    // tile.childDescriptors[i]...
                }
            }
        ]]>
    </mx:Script>

    <mx:Tile id="tile">
        <mx:TextInput id="input" />
        <mx:Button id="ok" label="OK" initialize="iterateOverChildren();" />
    </mx:Tile>
</mx:Application>
```

The following sections describe the properties of the `childDescriptors` property in more detail.

Using the `childDescriptors.id` property

The `id` property is the MXML ID of a control. Each MXML document has a flat namespace, so you must not use the same `id` property for two different components. If you do not specify an `id` property for a control, Flex generates an `id`, such as `_TextArea1`.

The following example shows that the `id` property of a `TextInput` control (index 0 of the `HBox` container), which does not specify an `id` property, results in `_TextInput1`. The `TextArea` control (index 1) results in `ta1`, because it has an `id` property.

```

<?xml version="1.0"?>
<mx:Application width='500' height='400'
  xmlns:mx="http://www.macromedia.com/2003/mxml" initialize="traceHB();" >

  <mx:HBox id="hb">
    <mx:TextInput/>
    <mx:TextArea id="ta1" />
  </mx:HBox>

  <mx:Script>
  <![CDATA[
    function traceHB() {
      trace(hb.childDescriptors[0].id) // Writes "_TextInput1"
      trace(hb.childDescriptors[1].id) // Writes "ta1"
    }
  ]]>
</mx:Script>
</mx:Application>

```

Using the `childDescriptors.properties` property

The `properties` property specifies the MXML properties of the specified control and its children. The following example shows the properties of a `TextArea` control in an `HBox` container:

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:Script>
  <![CDATA[
    function traceProps() {
      for (prop in hb.childDescriptors[0].properties) {
        trace(prop + " : " + hb.childDescriptors[0].properties[prop]);
      }
    }
  ]]>
</mx:Script>

  <mx:HBox id="hb">
    <mx:TextArea id="ta" initialize="traceProps();" preferredWidth="200"
      preferredHeight="25" text="hello ??" />
  </mx:HBox>
</mx:Application>

```

Using the `childDescriptors.type` property

The `type` property is the constructor function for the MXML tag that was used to create the control. This property is of type `Function`.

The following example shows the value of the `type` property:

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:HBox id="hb">
    <mx:TextInput/>
    <mx:Button id="ok" label="OK" click="traceHB();" />
  </mx:HBox>
</mx:Application>

```

```

</mx:HBox>

<mx:Script>
<![CDATA[
    function traceHB() {
        // Returns mx.controls.TextInput
        trace(hb.childDescriptors[0].type);

        // Returns mx.controls.Button
        trace(hb.childDescriptors[1].type);
    }
    ]]>
</mx:Script>
</mx:Application>

```

Setting container creation order

Flex lets you add any container to an instantiation queue. Once in the queue, Flex creates containers in the order in which they appear in the queue. However, Flex does not create the children of the container until all other queued containers are created. The result for large applications that have many containers is that the containers themselves appear on the screen and size themselves. In queue order, Flex then fills in the containers one at a time by creating all children of each container before creating the children of the next container.

The following image shows a complex application that contains a single container at the top right, and four containers across the bottom. During initialization, Flex first creates all the containers, and then fills in each container with its children and data. In this example, the image on the left shows the application after the first container is populated with its children and data, but before the remaining four containers are populated. The image on the right shows the final state of the application:



With an instantiation queue, you can specify that all views of a navigator container are created at startup, but that each view finishes before the next one begins. The result is that a user will see the first view while the remaining views continue to be instantiated.

This section describes how to add containers to the instantiation queue so that you can improve perceived layout performance of your Flex applications.

Adding containers to the queue

To add a container to the instantiation queue, set the value of the container's `creationPolicy` property to `queued`. You can add any number of containers to the queue. Unless you specify a queue order, containers are instantiated in the order in which they appear in the application source code.

In the following example, the panels are created in the order that they appear (panel1, panel2, panel3):

```
<?xml version="1.0" ?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:Panel id="panel1" creationPolicy="queued">
    <mx:Button id="button1a" />
    <mx:Button id="button1b" />
  </mx:Panel>

  <mx:Panel id="panel2" creationPolicy="queued">
    <mx:Button id="button2a" />
    <mx:Button id="button2b" />
  </mx:Panel>

  <mx:Panel id="panel3" creationPolicy="queued">
    <mx:Button id="button3a" />
    <mx:Button id="button3b" />
  </mx:Panel>
</mx:Application>
```

Flex creates all the containers first. Flex then instantiates all the children within each panel before moving on to instantiate the children within the next panel. The creation order for all controls in this application is as follows:

```
panel1
panel2
panel3
button1a
button1b
button2a
button2b
button3a
button3b
```

After Flex creates a container, Flex emits a `creationComplete` event for that container. Flex emits this event even if the container did not have any children. Flex then begins creating the children of the next container in the queue. The `creationComplete` event has a `type` and `target` property, corresponding to the type of event (`creationComplete`) and the name of the container that emitted the event.

Setting queue order

You can explicitly set the order in which containers are queued using the `creationIndex` property. Flex creates containers in their `creationIndex` order. If a container does not have a `creationIndex` set, it is created according to its `creationPolicy` property. By default, it is created before all other containers that do have a `creationIndex` property because the default value of `creationPolicy` for nonnavigator containers is `auto`.

All containers support a `creationIndex` property that is of type `Number`. If you set a `creationIndex` property on a container, but do not set the `creationPolicy` property to `queued`, Flex ignores the `creationIndex` property and uses the default value of `creationPolicy` for that container.

The following example creates the `Button` control in `HBox3` first, then `HBox2`, and finally `HBox1`:

```
<?xml version="1.0" ?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">

    <mx:VBox id="accord" width="300" height="200">
        <mx:HBox label="HBox1" creationPolicy="queued" creationIndex="3">
            <mx:Button label="Box 1" creationComplete="output.text += 'box1
created' + '\n'"/>
        </mx:HBox>

        <mx:HBox label="HBox2" creationPolicy="queued" creationIndex="2">
            <mx:Button label="Box 2" creationComplete="output.text += 'box2
created' + '\n'"/>
        </mx:HBox>

        <mx:HBox label="HBox3" creationPolicy="queued" creationIndex="1">
            <mx:Button label="Box 3" creationComplete="output.text += 'box3
created' + '\n'"/>
        </mx:HBox>

    </mx:VBox>
    <mx:TextArea id="output" width="200" height="200" />
</mx:Application>
```

If you set two `queued` containers to the same `creationIndex`, Flex creates them in the order in which they appear in the application. The `creationIndex` value can be any valid `ActionScript` `Number`, including 0 and negative values.

Dynamically adding containers to the queue

You can dynamically add containers to the instantiation queue using the `createLater()` method. All containers support this method. The `createLater()` method has the following signature:

```
function createLater ( id:String, preferredIndex:Number ):Void
```

The following table describes the `createLater()` method arguments:

| Argument | Description |
|-----------------------------|--|
| <code>id</code> | The <code>id</code> argument is the name of the container that you want to add to the queue. |
| <code>preferredIndex</code> | (Optional) The <code>preferredIndex</code> argument is the relative position in the queue to place the container. By default, Flex places the container at the end of the queue. |

You should only use the `createLater()` method to add containers to the queue whose `creationPolicy` is set to `none`. Containers whose `creationPolicy` is set to `auto` or `all` will most likely have already created their children during the application initialization. Containers whose `creationPolicy` is set to `queued` are already in the queue. The container that is to be created with the `createLater()` method then competes with containers whose `creationPolicy` is set to `queued`, depending on their position in the queue.

The following example uses the `createLater()` method to create the children of the `HBox` containers when the user clicks the Create Later button. The calls to the `createLater()` method specify a `preferredIndex` for each box, which causes the boxes to be created in a different order from the way they appear in the application (in this case, `box1`, `box3`, `box2`, and then `box4`):

```
<?xml version="1.0" ?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

    <mx:Script><![CDATA[
        function doCreate(){
            createLater('box1', 0);
            createLater('box2', 2);
            createLater('box3', 1);
            createLater('box4', 3);
        }
    ]]></mx:Script>

    <mx:HBox backgroundColor="#CCCCCC" horizontalAlign="center">
        <mx:Label text="CreateLater()" fontWeight="bold"/>
    </mx:HBox>

    <mx:Form>
        <mx:FormItem label="first:">
            <mx:HBox id="box1" creationPolicy="none" width="200" height="50"
borderStyle="solid" backgroundColor="#CCCCCC">
                <mx:Button label="Button 1" />
                <mx:Button label="Button 2" />
            </mx:HBox>
        </mx:FormItem>
        <mx:FormItem label="fourth:">
            <mx:HBox id="box2" creationPolicy="none" width="200" height="50"
borderStyle="solid" backgroundColor="#CCCCCC">
                <mx:Button label="Button 3" />
                <mx:Button label="Button 4" />
            </mx:HBox>
        </mx:FormItem>
        <mx:FormItem label="second:">
```

```

        <mx:HBox id="box3" creationPolicy="none" width="200" height="50"
borderStyle="solid" backgroundColor="#CCCCFF">
            <mx:Button label="Button 5" />
            <mx:Button label="Button 6" />
        </mx:HBox>
    </mx:FormItem>
    <mx:FormItem label="third:">
        <mx:HBox id="box4" creationPolicy="none" width="200" height="50"
borderStyle="solid" backgroundColor="#CCCCFF">
            <mx:Button label="Button 7" />
            <mx:Button label="Button 8" />
        </mx:HBox>
    </mx:FormItem>
</mx:Form>

    <mx:Button label="Create Later" click="doCreate();" />
</mx:Application>

```

In some cases, the `createLater()` method may not act as you might expect. The reason for this is that if the instantiation queue is empty, the first container to be put in that queue triggers the creation process of its children. However, that item might not have the lowest `preferredIndex` value.

In the following example, although `box1` has the highest `preferredIndex`, Flex creates its children first. By the time `box1` is complete, the other containers will be in the queue, and Flex proceeds with the next lowest item in the queue.

```

function doCreate(){
    createLater('box1', 4);
    createLater('box2', 2);
    createLater('box3', 1);
    createLater('box4', 3);
}

```

The creation order for this example is as follows:

```

box1
box3
box2
box4

```

Combining containers with different `creationPolicy` settings

You can mix containers with different `creationPolicy` settings and reorder their instantiation order. Outer containers are created before inner containers, regardless of their `creationIndex`. This is because Flex does not create the children of a queued container until all containers at that level are created.

The following pseudocode example specifies a `creationIndex` order for all box containers:

```

box1 (creationIndex = 6)
box2 (creationIndex = 1)
box3 (creationIndex = 5)
button1
box4 (creationIndex = 2)
button2

```

```
box5 (creationIndex = 3)
button3
button4
box6 (creationIndex = 4)
button5
```

The following is the order in which Flex creates the components. Components created at the same time are on the same line

```
box1
box2, button4, box6
box3, box5
button3
button5
button1, box4
button2
```

The following shows the contents of the queue, over time:

```
Q: box1
createComponents called on box1, creating box 2, button4, & box6
Q: box2, box6
createComponents called on box2, creating box3 & box5
Q: box5, box6, box3
createComponents called on box5, creating button3
Q: box6, box3
createComponents called on box6, creating button5
Q: box3
createComponents called on box3, creating button1 & box4
Q: box4
createComponents called on box4, creating button2
Q: <empty>
```

Setting a container's `creationPolicy` to `queued` does not override the policies of the containers within that container. As a result, if you queue an outer container, but set the inner container's `creationPolicy` to `none`, Flex creates the inner container, but not any child controls of that inner container unless you explicitly call the `createComponent()` or `createComponents()` methods on that inner container.

In the following example, the `button1` control is never created because its container specifies a `creationPolicy` of `none`:

```
<?xml version="1.0" ?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">

    <mx:HBox label="HBox1" creationPolicy="queued" creationIndex="0">
        <mx:HBox label="HBox1" creationPolicy="none">
            <mx:Button id="button1" label="Button 1"
                creationComplete="output.text += 'button1 created' + '\n' " />

        </mx:HBox>
    </mx:HBox>

    <mx:HBox label="HBox2" creationPolicy="queued" creationIndex="1">
        <mx:Button id="button2" label="Button 2" creationComplete="output.text +=
            'button2 created' + '\n' " />
    </mx:HBox>
</mx:Application>
```



```

    </mx:HBox>
    <mx:TextArea id="output" width="200" height="200" />
</mx:Application>

```

Using effects with queued containers

When you set the `creationPolicy` property to `queued` for a set of containers, you can use the `childrenCreationCompleteEffect` effect trigger to play an effect just as the children of each container become visible. The `childrenCreationCompleteEffect` trigger is paired with the `childrenCreationComplete` event. After a piece of the application is initialized and appears, Flex waits for all effects to finish playing before initializing the next piece.

In the following example, the `childrenCreationCompleteEffect` effect trigger is set to the `Dissolve` effect for two queued `Panel` containers:

```

<mx:Application creationCompleteEffect="Fade">
  <mx:Panel width="200" height="100"
    creationPolicy="queued"
    childrenCreationCompleteEffect="Dissolve">
    ...
  </mx:Panel>
  <mx:Panel width="200" height="100"
    creationPolicy="queued"
    childrenCreationCompleteEffect="Dissolve">
    ...
  </mx:Panel>
</mx:Application>

```

When the application is requested, the following sequence of events occurs:

1. The `Application` container and the two `Panel` containers, but not their children, initialize.
2. The `Panel` containers fade into view, using a `Fade` effect.
3. After the `Fade` effect finishes, the children of the first `Panel` container appear, using a `Dissolve` effect.
4. After the `Dissolve` effect for the first `Panel` container finishes, the children of the second `Panel` container appear, using a `Dissolve` effect.

Setting creation order with asynchronous events

Sometimes the developer needs to perform some specialized logic or processing before creating the children of Flex containers. This is often the case when the container's content is dynamically generated based on data from an asynchronous service such as a `WebService`, `HTTPService`, or `RemoteObject`.

You can use the `createLater()` method to dynamically add containers to the end of the queue. With this method, set the `creationPolicy` of the target containers to `none`, and call `createLater()` on those containers only when the results of the service are returned.

The following example gets data from two asynchronous DataProviders; one is a WebService and the other is an HTTPService. When the services return data to the Flex application, Flex calls the `createLater()` method and adds the container to the end of the queue. It does not matter which container gets its data first, because Flex begins creation of that container's children when the first item is added to the instantiation queue. Ideally, the second service will return its data before the first container is finished being created so that the staggered population of data is transparent to the user.

```
<?xml version="1.0" ?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:Script><![CDATA[
    function getServices() {
      wsl.getData();
      hsl.send();
    }
  ]]></mx:Script>

  <mx:WebService id="wsl" wsdl="http://somewhere.com/my.wsdl">
    <mx:operation name="getData" result="createLater('panel1')"/>
  </mx:WebService>

  <mx:HTTPService id="hsl" url="data.xml" method="POST"
    result="createLater('panel2')/>

  <mx:HBox id="b1" DONEEVENT="getServices()">
    <mx:Panel id="panel1" creationPolicy="none">
      <mx:DataGrid id="dgl" dataProvider="{wsl.result}"/>
    </mx:Panel>
    <mx:Panel id="panel2" creationPolicy="none">
      <mx:Tree id="tree1" dataProvider="{hsl.result}"/>
    </mx:Panel>
  </mx:HBox>
</mx:Application>
```

CHAPTER 25

Using Runtime Shared Libraries

Macromedia Flex supports Runtime Shared Libraries (RSLs). This chapter describes how to configure and use them to take advantage of their benefits.

Contents

| | |
|---|-----|
| About RSLs | 595 |
| Using RSLs | 597 |
| Creating RSLs for precompiled Flex applications | 605 |
| RSL errors | 606 |

About RSLs

When Flex compiles an MXML file and its assets into a SWF file, the resulting SWF file is a single entity, consisting of all the base application model components (such as Button, CheckBox, and Panel), plus graphical assets, embedded data, and custom components. The result can be a large file that can be slow to download. In many cases, Flex implementations have multiple applications that share some of the same assets. But clients were required to download the entire application and all of the components for each Flex application they accessed.

You can now reduce the size of your application's SWF file by externalizing shared assets into standalone files that can be separately downloaded and cached on the client. These shared assets are loaded by any number of applications at runtime, but only need to be transferred to the client once. These shared files are known as *Runtime Shared Libraries* or *RSLs*.

If you have multiple applications but those applications share a core set of graphic files, components, and other assets, your users only have to download those assets once in an RSL. The resulting file size for your main application can be dramatically reduced.

If a file in one of the RSLs changes, Flex recompiles and resends that RSL to the client. The application and other unchanged RSLs are not recompiled.

What is an RSL?

An RSL is a SWF file that contains Flex assets such as images, components, or other SWF files. The SWF file itself is actually extracted from a SWC file before being sent to the client, where it is cached locally by Macromedia Flash Player. Components and assets defined within the RSL stay in this cached file. The next time the Flex application runs, Flash Player loads the RSL from the local client's cache. If the RSL descriptor file or its dependencies change, Flash Player gets a new SWF file.

Flex generates RSL SWF files at runtime based on the settings of your Flex application. The following are the most common types of RSLs:

- A package of custom components
- Commonly used Macromedia components
- Large sets of images or application branding assets
- Some combination of the above

The following applications can benefit from RSLs:

- Large applications that load multiple smaller applications that are linked to a shared RSL. A Flex application that contains other, nested Flex applications. The Flex Explorer sample application is an example of this. In this scenario, the top-level application and all the subordinate applications can share components that are stored in a common RSL.
- A family of applications on a server built with a shared RSL. When the user visits the first application, they download an application SWF and a shared RSL. When they visit the second application, they download only the application SWF (the components in the RSL are shared between the two applications).
- A frequently changed application that has a large set of infrequently changed components.
- An infrequently changed application that has frequently changed data or assets. For example, a small piece of an application is dynamically generated, using a JSP or ASP.NET file that emits MXML. Most of the application is static. If the static pieces are packaged in an RSL, Flex must only recompile and send a small piece of the application each time.

Considerations

Using RSLs is not the right solution for everyone and it does not solve all problems of application performance. Using RSLs has the following disadvantages:

- The aggregate size of the RSL plus the Flex application will be larger than a Flex application that internalizes all of its assets. However, file size across multiple applications will be smaller.
- The client must make additional HTTP requests to get the RSLs used by a Flex application. This can add latency to the application startup process. Typically, the number of RSLs is not great, so the number of requests is not great, either.
- The compilation of a runtime RSL can add additional application startup time because a separate compiler instance must be created on the server. This adds latency to the first request only, since subsequent requests will request the existing RSL without the need to recompile it.

Using RSLs

There are two aspects to using RSLs: defining their contents and adding them to your application.

Before you can use an RSL in your application, you must define it. To define an RSL, you create a SWS file (with an *.sws extension) that lists the resources contained in the RSL, as the following example shows:

```
<library>
  <namespace uri="http://www.macromedia.com/2003/mxml">
    <component name="Button" />
  </namespace>
  <embed source="images/icon1.jpg" />
  <embed source="myTheme.swc" symbol="cursor" />
</library>
```

You store the SWS file in any web-accessible directory. After you define the contents of the RSL, it is very simple to add the RSL to your application. You add an `rsl` attribute in the

`<mx:Application>` tag that points to the SWS file. For example:

```
<mx:Application rsl="myLib.sws">
```

On the first request for an application that uses that RSL, Flex compiles the RSL using the SWS file's list of contents. If you change the assets in the RSL or the application itself, Flex recompiles the application by default. However, you can instruct Flex to not recompile the application when you change the RSL assets.

The following sections describe the process of adding RSLs to your applications in detail.

Defining the contents of an RSL

To define the contents of an RSL, you create a SWS file that lists components, classes, and other assets to be included in the RSL. This XML file is known as the *library descriptor file*; it provides a convenient location to track all the members of a particular RSL and lets you easily add and remove assets.

When a client first requests a Flex application that references a library descriptor file, Flex compiles a SWC file out of the assets listed in that file. This compilation only takes place once, and you can precompile a SWC file to avoid the initial compile time.

Flex then extracts the implementing SWF file from the library SWC file, and returns both the Flex application's SWF and the RSL's SWF file to the client. The client locally caches the contents of the RSL SWF file. Subsequent requests that rely on assets listed in the library descriptor file do not require downloading those assets since they are already stored on the client.

Assets can include images, sounds, SWC files, and SWF files. The contents of an RSL can be any file type that is supported by the `Embed` metadata keyword. For more information about the `Embed` metadata keyword, see [“Importing images using Embed in ActionScript” on page 293](#).

When an asset is added to an RSL, Flex adds all of that asset's dependencies to the RSL as well. You are not required to explicitly include all dependencies for that asset. For example, if you add the Button control to an RSL, Flex adds the `mx.core.UIComponent`, `mx.core.UIObject`, and `mx.controls.SimpleButton` controls to the RSL because the Button control derives from those classes.

At first this might sound like it would slow the download process for a Flex application, however, remember that after the first time a Button component RSL is downloaded, all subsequent applications that use the Button control do not have to compile or send these classes to the client.

The following section describes the syntax of the library descriptor file.

Library descriptor file syntax

The top-level tag of the library descriptor file is `<library>`. There are also several child tags that you can use in a library descriptor file. The following table describes the tags and attributes you can use in a library descriptor file:

| Tag | Description |
|--|---|
| <pre><library debug="true false" lib="rsl_or_swf" preload="true false" rebuildClients="true false" root="root_path" rsl="rsl_or_swf" url="location_of_SWF" ></pre> | <p>Defines RSL properties for the application.</p> <p>(Optional) Set <code>debug="true"</code> to include debugging information in the RSL. Flex adds the SWD file to the RSL's SWC file, and subsequently extract the SWD file to return to the client. The default is <code>false</code>.</p> <p>(Optional) Set the <code>lib</code> attribute to point to other library descriptor files or SWC files that are statically linked. Using this attribute adds the resources in this library descriptor file to the single, statically linked application file rather than an RSL. The usage for this attribute is the same as the usage for the <code>lib</code> attribute on the <code><mx:Application></code> tag.</p> <p>(Optional) Set <code>preload="false"</code> to instruct the application preloader to skip this RSL. The default is <code>true</code>.</p> <p>(Optional) Set <code>rebuildClients="true"</code> to trigger Flex to recompile the application when files in the library change. Setting this attribute to <code>false</code> prevents Flex from recompiling the application when the library changes. This can result in faster compile times. However, setting this to <code>false</code> can also result in persistent compiler errors if Flex encounters a problem. The default is <code>true</code>.</p> <p>(Optional) Set the <code>root</code> attribute to the relative location that you want RSL assets to be accessible from in your Flex applications. Otherwise, the application root of the RSL assets defaults to the location of the library descriptor file. For more information, see "Specifying the root for RSL assets" on page 605.</p> <p>(Optional) Set the <code>rsl</code> attribute to point to other library descriptor files or SWC files to be linked at runtime. The usage for this attribute is the same as the usage for the <code>rsl</code> attribute on the <code><mx:Application></code> tag. For more information, see "Using RSLs in your applications" on page 604.</p> <p>(Optional) Set the <code>url</code> attribute when you are precompiling your MXML files into SWF files before deploying them on your server. The value is the location of the SWF file. For more information, see "Creating RSLs for precompiled Flex applications" on page 605.</p> |
| <pre><namespace uri="uri" all="true false"></pre> | <p>Begins a block of components to include in the RSL.</p> <p>(Required) Specify a <code>uri</code> attribute to identify a component namespace for all components in this block. The value of the <code>uri</code> attribute can be a full URI or a global namespace such as <code>""</code>.</p> <p>(Optional) Set <code>all</code> to <code>true</code> to load all components in the entire namespace, just as if you had entered all the <code><component></code> tags individually. The default is <code>false</code>.</p> |

| Tag | Description |
|--|--|
| <pre><component name="component_name" uri="uri" ></pre> | <p>Adds a single component to the RSL. The <code><component></code> tag can be a child tag of the <code><namespace></code> tag or a stand-alone tag.</p> <p>(Required) The <code>name</code> attribute specifies the local name of the component.</p> <p>The <code>uri</code> attribute identifies a component namespace. The value of the <code>uri</code> attribute can be a full URI or a global namespace, such as <code>"*"</code>. This attribute is optional if the <code><component></code> tag is nested within a <code><namespace></code> tag. This attribute is required if the <code><component></code> tag is not a child tag of the <code><namespace></code> tag.</p> |
| <pre><embed mimeType="MIME_type" newSymbol="linkage_id_out" source="source_file"] symbol="linkage_id_in" ></pre> | <p>Adds an asset to the RSL. Assets can include images, sounds, SWF files, and symbols in SWF or SWC files.</p> <p>(Optional) The <code>mimeType</code> attribute specifies a MIME type for the asset. This attribute is only necessary if the filename's extension is ambiguous or nonexistent, or if Flex cannot detect the MIME type from the file's headers.</p> <p>(Optional) The <code>newSymbol</code> attribute is a name that you use to refer to the symbol in your Flex application.</p> <p>The <code>source</code> attribute specifies the source file for an asset. This attribute is optional if the source file of the asset or the symbol can be found implicitly. For example, if the asset is a symbol inside a SWF file and the SWF file is in the <code>user_classes</code> directory, Flex can find and extract the symbol without a <code>source</code> attribute. This attribute is required if you do not specify a <code>symbol</code> attribute.</p> <p>The <code>symbol</code> attribute is the linkage ID or symbol name of the asset in the source file. This attribute is required only if you are adding a symbol asset to the RSL.</p> |

Adding components to a library descriptor

The most common use of RSLs is to add commonly used components. You use the `<component>` child tag of the `<namespace>` tag to add individual components to your RSL. The following example adds a subset of the Flex frameworks components to the RSL:

```
<library>
  <namespace uri="http://www.macromedia.com/2003/mxml">
    <component name="Button" />
    <component name="VBox" />
    <component name="HBox" />
    <component name="TextInput" />
    <component name="Application" />
    <component name="Panel" />
  </namespace>
</library>
```

The `<component>` child tag can also take a `uri` attribute that defines the component's namespace. The following example adds the `MyButton` custom component to the library descriptor:

```
<library rebuildClients="true">
  <component name="MyButton" uri="*" />
</library>
```


In your Flex code, you must still declare the namespace for a custom component. In the following example, the global namespace URI indicates that the custom component is in the same directory as the Flex application file:

```
<MyButton xmlns="*" click="reOrder();" />
```

If the custom component is not in the same directory as the Flex application, you use the `root` attribute of the `<library>` tag to specify a directory that Flex can then find the component from. The Flex compiler can find sources such as ActionScript class files and MXML component files if they are in the ActionScript classpath or in the `user_classes` directory. These kinds of sources are ignored if they are in the `lib-path`.

RSLs that include components with dependencies also include those dependencies by default. You are not required to add all component dependencies in the library descriptor file.

When you use a `<namespace>` tag without any `<component>` child tags, you can add the `all="true"` attribute so that Flex adds all components in that namespace URI to the RSL, as long as that namespace has a manifest file associated with it.

The following example adds all components in the Flex framework to your RSL:

```
<library>
  <namespace uri="http://www.macromedia.com/2003/mxml" all="true" />
</library>
```

Manifest files list the components in a namespace. An example is the Macromedia MXML namespace. The manifest file found in the *flex_app_root*/WEB-INF/flex/frameworks/mx.swc file explicitly defines all the components in that namespace.

In the *flex-config.xml* file, you map a URI to a manifest file using the `<namespaces>` tag, as the following example shows:

```
<namespaces>
  <namespace uri="http://www.macromedia.com/2003/mxml">
    <manifest>/WEB-INF/flex/mxml-manifest.xml</manifest>
  </namespace>
</namespaces>
```

The manifest file lists the components. For more information about manifest files, see [“Using manifest files” on page 805](#).

Adding assets to a library descriptor

You can add assets such as graphics, SWF files, and sounds to an RSL. To do this, you use the `<embed>` child tag of the `<library>` tag in the library descriptor file.

The following example adds a small set of images to an RSL:

```
<library>
  <embed source="images/icon1.jpg" />
  <embed source="images/icon2.jpg" />
  <embed source="images/icon3.jpg" />
  <embed source="images/icon4.jpg" />
</library>
```

In your Flex application, you refer to those assets with the `<mx:Image>` tag that use the `@Embed` syntax, as the following example shows:

```
<mx:Canvas id="canvas1">
  <mx:Image id="i1" source="@Embed('images/icon1.jpg')" x="10" y="10" />
  <mx:Image id="i2" source="@Embed('images/icon2.jpg')" x="10" y="50" />
  <mx:Image id="i3" source="@Embed('images/icon3.jpg')" x="10" y="90" />
  <mx:Image id="i4" source="@Embed('images/icon4.jpg')" x="10" y="130" />
</mx:Canvas>
```

Similarly, if you add sounds, SWF files, or other assets, you use the `Embed` metadata keyword or other Flex tag, just as you would normally embed those assets. For more information, see [“Importing images using Embed in ActionScript” on page 293](#).

In some cases, you might want to rename an asset’s symbol name so that you can refer to it in your applications with a different name. This gives you more control over how the application developer refers to those assets in the Flex application, and can help avoid naming collisions if the asset’s original name is common or simple. It also lets you hide long, unwieldy asset source strings if you distribute your RSLs as part of a component library.

To rename a symbol, you use the `newSymbol` attribute of the `<embed>` tag in the library descriptor. The `newSymbol` attribute defines the name that the application developer can then use to refer to the asset in their Flex applications.

The following example library descriptor file defines the symbol names for the images, in addition to specifying the source of those assets:

```
<library>
  <embed source="images/icon1.jpg" newSymbol="CompanyLogoLarge" />
  <embed source="images/icon2.jpg" newSymbol="CompanyLogoExtraLarge" />
  <embed source="images/icon3.jpg" newSymbol="CompanyLogoSmall" />
  <embed source="images/icon4.jpg" newSymbol="CompanyLogoTiny" />
</library>
```

In your Flex application, you refer to those assets with the `@Embed` syntax of the `<mx:Image>` tag and point to the symbol name rather than the image name, as the following example shows:

```
<mx:Canvas id="canvas1">
  <mx:Image id="i1" source="@Embed(symbol='CompanyLogoLarge')" x="10" y="90"/>
  <mx:Image id="i2" source="@Embed(symbol='CompanyLogoExtraLarge')" x="10"
  y="90"/>
  <mx:Image id="i3" source="@Embed(symbol='CompanyLogoSmall')" x="10" y="90"/>
  <mx:Image id="i4" source="@Embed(symbol='CompanyLogoTiny')" x="10" y="90"/>
</mx:Canvas>
```

Adding embedded symbols to a library descriptor

Embedded symbols are assets stored inside a SWC or SWF file that you can extract and add to your applications. You can add symbols that are embedded in SWC and SWF files to an RSL using the `<embed>` child tag in the library descriptor file.

You should only embed assets that do not have any class dependencies, such as images.

The following example adds the cursor symbol from the myTheme.swc file to the RSL:

```
<library>
  <embed source="myTheme.swc" symbol="cursor" />
</library>
```

As with other assets, you can use the @Embed syntax to refer to embedded symbols in the Flex application code, as the following example shows:

```
<mx:Image id="image5" source="@Embed('myTheme.swc#cursor')" />
```

Also, as with other assets, you can use the newSymbol attribute to define a new name that you can use to refer to the embedded symbol. The following sample line from a library descriptor adds the cursor symbol and gives the symbol the name *NewLogo*:

```
<embed source="myTheme.swc" symbol="cursor" newSymbol="NewLogo" />
```

An application developer would then refer to this symbol in their code in the following way:

```
<mx:Image id="logo" source="@Embed(symbol='NewLogo')" />
```

You are not required to specify the source attribute of the <embed> tag if you are extracting a symbol from a SWC file. As long as the symbol's SWC file is in the *flex_app_root*/WEB-INF/flex/user_classes directory or other directory specified by the <lib-path> tag in the flex-config.xml file, Flex detects the source of the symbol and adds it to the RSL.

For example, if you have the MyGraphics.swc file in your user_classes directory, and this file contains a symbol called CompanyLogo, you can add the CompanyLogo symbol to your RSL with the following line in your library descriptor:

```
<embed symbol="CompanyLogo" />
```

For more information about the <lib-path> element in the flex-config.xml file, see [“Editing the flex-config.xml file” on page 806](#).

Chaining RSLs

You can chain RSLs to include other RSLs by using the rsl attribute of the <library> tag in the SWS file. This lets you group resources for easier reusability.

The following example defines MyRSL1 that includes four images. In addition, it includes a second RSL.

MyRSL1.sws:

```
<library rsl="MyRSL2.sws">
  <embed source="images/icon1.jpg" newSymbol="CompanyLogoLarge" />
  <embed source="images/icon2.jpg" newSymbol="CompanyLogoExtraLarge" />
  <embed source="images/icon3.jpg" newSymbol="CompanyLogoSmall" />
  <embed source="images/icon4.jpg" newSymbol="CompanyLogoTiny" />
</library>
```

MyRSL2.sws:

```
<library>
  <embed source="images/icon5.jpg" newSymbol="OtherCompanyLogoLarge" />
  <embed source="images/icon6.jpg" newSymbol="OtherCompanyLogoExtraLarge" />
  <embed source="images/icon7.jpg" newSymbol="OtherCompanyLogoSmall" />
</library>
```

```
<embed source="images/icon8.jpg" newSymbol="OtherCompanyLogoTiny" />
</library>
```

Do not include the same resource in two different SWS files.

Using RSLs in your applications

To use an RSL in your Flex application, specify a value for the `rsl` attribute in the `<mx:Application>` tag that points to the library descriptor file or to a SWC file. The location is relative to the application root, or you can use an absolute path. If you begin the path with a forward slash (`/`), the location of the file is relative to the server root. The library descriptor file and its contents can be in any location that is web accessible.

The following example instructs the current application to use the `myLib.sws` file as the library descriptor file:

```
<mx:Application rsl="libs/myLib.sws">
    ...
</mx:Application>
```

You can only specify the `rsl` attribute in the root `<mx:Application>` tag of a Flex application. You can specify multiple library files by separating them with semi-colons, as the following example shows:

```
<mx:Application rsl="myLib.sws;myOtherLib.sws">
```

You can also use prebuilt SWC files as RSLs. In this case, point the `rsl` attribute to the SWC file, as the following example shows:

```
<mx:Application rsl="mylib/foo.swc">
```

As with all SWC files, you should not store RSL SWC files in the application root directory. However, you are not required to store an RSL SWC in the `user_classes` directory or other directory specified by the `<lib-path>` tag in the `flex-config.xml` file. Flex finds RSL SWC files using the path specified with the `rsl` attribute on the `<mx:Application>` tag.

Virtual directories might produce a warning about being unable to compute an absolute URL. In these cases, you must manually override the URL in the library descriptor file.

Precompiling SWC files

You can precompile the RSL SWC files for your application using the `compc` command-line utility. You then specify the name of the compiled SWC file as the argument to the `rsl` attribute, as the following example shows:

```
<mx:Application rsl="myLib.swf.swc">
```

By specifying a SWC file rather than a SWS file, you speed up the RSL process. Flex is no longer required to compile the SWC file, it only needs to extract the SWF file from the SWC file to send to the client.

The default output directory for a SWC file generated from a SWS file is *flex_app_root/WEB-INF/flex/generated/hash-map/library_name.swc* file. You can override this directory with the `genlibdir` option. When you precompile an RSL for an application that will later precompile with `mxmcl` and deploy on another server, you must override the output directory and make it match the target directory within the deployed application.

For more information on using `compc`, see [“Using SWC files” on page 800](#).

Specifying the root for RSL assets

If you use the `source` attribute when referring to an asset (for example, in the `<mx:image>` tag), the location of the asset is relative to the location of the RSL's library descriptor file if that asset is within an RSL. If the asset is not in an RSL, then the relative location of that asset is based on the application root. If your application and RSL are in the same directory, then the root location of all assets is the same.

You can set the relative location of the RSL's root by using the `root` attribute of the `<library>` tag in the library descriptor file. Otherwise, if the RSL and the application are not in the same directory, you must make sure the source path in your `<mx:image>` tag points to the correct location (the application root or the RSL root, depending on whether the asset is in the RSL or not). Alternatively, you can store the asset in two different directories.

The Flex compiler will find any type of resource in the `user_classes` directory.

If you use virtual directories in your web application, Flex might raise an error about being unable to compute absolute URLs. You might be required to manually override the URL in the library descriptor file.

Creating RSLs for precompiled Flex applications

When you deploy your applications, you can either deploy them as MXML files or precompile them using the `mxmcl` compiler.

There are no special considerations for using RSLs when deploying applications as MXML files. However, precompiling Flex applications presents a special problem. Because you precompile, it is likely that the location of the RSLs compiled into the application will not match your final deployed location. Therefore, you must specify the location of the RSL library's SWF file, so that it can be included in the application at compile-time.

In addition, if the location of your RSL is in a virtual directory, Flex might generate warnings that it cannot find the RSL when it attempts to generate an absolute URL. You can manually override the virtual directory with a URL using the technique in this section.

To use the RSLs with precompiled Flex applications:

1. Set the value of the `url` attribute of the `<library>` tag to point to the location of your RSL, as the following example shows:

```
<library url="http://localhost:8101/devlibs/MyLibrary.swf">
```

2. Generate the RSL SWC file using one of the following methods:

- Use the `compc` command-line compiler. When you use `compc`, you choose the output location of the SWC file. For example:

```
compc -o ../mylib/MyLibFile.swc -root .. /mylib
```

You should not store RSL SWC files in the application root directory. For more information, see [“Using SWC files” on page 800](#).

- Request the application that uses the RSL. Flex generates the SWC file for the RSL in the `flex_app_root/WEB-INF/flex/generated/hash-map/library_name.swc` file.

3. Open the RSL SWC file in WinZip or a similar archiving utility.
4. Extract the `library.swf` file from the SWC file. If the RSL contains only a single component, the name of the SWF file is the same name as the component.
5. Change the name of the `library.swf` file to the name you specified in the `url` attribute in step 1.. For example, rename the `library.swf` file to *MyLibrary.swf*.
6. Compile your Flex application.

RSL errors

Errors can occur when you use RSL files with your Flex applications. The following are the most common errors related to RSLs:

- RSL failed to compile.
- RSL file could not be found on the server.
- Network failure prevented the RSL from being transferred to the client.

If Flex cannot compile the RSL, it is possible that one or more of the components or assets specified in the library descriptor file cannot be found. Check that the asset’s source file exists and that the location specified in the library descriptor file is accurate.

If the asset is stored in a SWC file, be sure to either refer to the SWC file explicitly using the `embed source` attribute or make sure that the SWC file is in your `libpath`. You do this by adding the resource’s SWC file to the `flex_app_root/WEB-INF/flex/user_classes` directory or to a location that is specified by the `<lib-path>` element in the `flex-config.xml` file.

If you get an error message such as “Unable to resolve library locations”, Flex cannot find the library descriptor file that is specified in the `<mx:Application>` tag. Check the path and filename of the library descriptor file.

Setting the `rebuildClients` attribute of the `<library>` tag to `false` in the library descriptor file can cause persistent compiler errors. If the RSL experiences an error, the result is that Flex does not recompile the application or the RSL, and the existing RSL does not function properly. Therefore, when you test applications that use RSLs, you should set `rebuildClients` to `true` (the default).

CHAPTER 26

Printing from SWF Files

Many Macromedia Flex applications let users print from within the application. Users may want to print an entire Macromedia Flash Player screen, or they might only want to print parts of the screen. For example, you might have an application that returns confirmation information after a user completes a purchase. You can build functionality into your application that lets users print that page to keep for their records.

This chapter describes the options for printing all or parts of a Flex application.

Contents

| | |
|---|-----|
| About Printing. | 607 |
| Printing from the Flash Player context menu | 608 |
| Using the ActionScript PrintJob class. | 609 |
| Starting a print job. | 612 |

About Printing

You can add printing functionality to your applications that lets users print from Flash Player. To print an entire application screen, users access the Flash Player context menu and select the Print command.

To add greater control over printing than provided by the Flash Player context menu, or to build a print option directly into your user interface, use the ActionScript PrintJob class. This class gives you control over how the user prints the application or individual components of the application. You can print the component as it is currently displayed on the screen, or you can add logic to modify the component to optimize it for printing.

Note: The API reference for the PrintJob class is contained in *Flex ActionScript Language Reference*.

Additionally, users can print from a browser, rather than from Flash Player, by selecting a command such as File > Print from the browser window.

Advantages of printing from Flash Player

Printing from Flash Player directly, rather than from a browser window Print menu, offers the following advantages:

- Users can print all of the application, or only some parts of the application.
- You can specify that the content prints as vector graphics (to take advantage of higher resolution) or as bitmaps (to preserve transparency and color effects).
- The `PrintJob` class adds the ability to print dynamically rendered pages as a single print job. The `PrintJob` class also provides the user's printer settings, which you can use to format reports specifically for the user. For more information, see [“Using the ActionScript PrintJob class” on page 609](#).

Supported printers

With Flash Player, you can print to PostScript and non-PostScript printers. For a list of supported Flash Player printing platforms, see the “Macromedia Flash Player Web Printing FAQ” on the Macromedia website (www.macromedia.com/software/flash/open/webprinting/faq.html).

Printing from the Flash Player context menu

The simplest way to print a Flex application is to use the Print command in the Flash Player context menu. While your Flex application is executing, you open the context menu by right-clicking (Windows) or control-clicking (Macintosh) in Flash Player.

The context menu's Print command cannot print transparency or color effects. Instead, use the `PrintJob` class. For more information, see [“Using the ActionScript PrintJob class” on page 609](#).

To print frames using the Flash Player context menu Print command:

1. Start your Flex application.
2. Right-click (Windows) or Control-click (Macintosh) in Flash Player in the browser window to display the Flash Player context menu.
3. Select Print from the Flash Player context menu to display the Print dialog box.
4. In Windows, select the print range to select which frames to print:
 - Select All to print the entire application.
 - Select Pages and enter a range to print the labeled frames in that range.
 - Select Selection to print the current frame.
5. On the Macintosh, in the Print dialog box, select the pages to print:
 - Select All to print the current frame if no frames are labeled or to print all labeled frames.
 - Select From and enter a range to print the labeled frames in that range.
6. Select other print options, according to your printer's properties.
7. Click OK (Windows) or Print (Macintosh).

Note: Printing from the context menu does not interact with calls to the `PrintJob` class.

Using the ActionScript PrintJob class

The ActionScript `PrintJob` class lets you print an entire application screen, an individual component, or multiple components. You can use the `PrintJob` class to render dynamic content at runtime, prompt users with a single print dialog box, and print an unscaled document with proportions that map to the proportions of the content. This capability is especially useful for rendering and printing external dynamic content, such as database content and dynamic text.

Additionally, with properties populated by the `PrintJob.start()` method, your document can access your user's printer settings, such as page height, width, and orientation, and you can configure your document to dynamically format Flash content that is appropriate for the printer settings.

Often, you use the `PrintJob` class within an event handler. For example, you can add a `Button` control to your application that, when selected, prints some or all of the application from within the `Button` control's event handler.

Building a print job

To build a print job, you use functions that complete the tasks in the order described later in this section. The sections that follow the procedure provide explanations of the functions and properties associated with the `PrintJob` class.

Because you are spooling a print job to the user's operating system between your calls to the methods `PrintJob.start()` and `PrintJob.send()`, and because the `PrintJob` functions might temporarily affect the Flash Player internal view of onscreen Flash content, you should implement print-specific activities only between your calls to the methods `PrintJob.start()` and `PrintJob.send()`. For example, the Flash content should not interact with the user between the methods `PrintJob.start()` and `PrintJob.send()`. Instead, you should expeditiously complete the formatting of your print job, add pages to the print job, and send the print job to the printer.

To build a print job:

1. Create an instance of the `PrintJob` class: `new PrintJob()`.
2. Start the print job and display the print dialog box for the operating system:
`PrintJob.start()`.
3. Add pages to the print job (call once per page to add to the print job): `PrintJob.addPage()`.
4. Send the print job to the printer: `PrintJob.send()`.
5. Delete the print job: `delete PrintJob`.

Only one print job may can run at any given time. You cannot start a second print job until one of the following has happened with the previous print job:

- The print job was entirely successful and the `PrintJob.send()` method was called.
- The `PrintJob.start()` method returned a value of `false`.
- The `PrintJob.addPage()` method returned a value of `false`.
- The `delete PrintJob` method was called.

Following is an `ActionScript` example that creates a print job for a `DataGrid` control:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">

    <mx:Script>
        <![CDATA[

            // Create a PrintJob instance.
            function doPrint() {
                var pj : PrintJob = new PrintJob();

                // Start the print job.
                if(pj.start() != true){
                    delete pj;
                    return;
                }

                // Add pages.
                pj.addPage(myDataGrid);
                pj.send();
                // Delete print job.
                delete pj;
            }
        ]]>
    </mx:Script>

    <mx:HBox>
        <mx:Button id="myButton" label="Print" click="doPrint()" />
        <mx>DataGrid id="myDataGrid" width="300" height="200" >
            <mx:DataProvider>
                ...
            </mx:DataProvider>
        </mx>DataGrid>
    </mx:HBox>
</mx:Application>
```

In this example, selecting the `Button` control invokes the `doPrint` event handler. The event handler creates an instance of the `PrintJob` class to print the `DataGrid` control, adds the `DataGrid` control to the print job using the `addPage()` method, and then uses the `send()` method to print the page.

By default, Flex prints the entire `DataGrid` control. If you want only to print part of the control, you can specify a second argument to the `addPage()` method that defines the print area of the control. The following example prints only the top half of the `DataGrid` control:

```
var dgPrintArea={xMin:0,xMax:myDataGrid.width,yMin:0,yMax:myDataGrid.height/2}
pj.addPage(myDataGrid,dgPrintArea);
```

The `xMin`, `yMin`, `xMax`, and `yMax` values are relative to the upper-left corner of the control. For more information, see [“Specifying a print area” on page 613](#).

You can also print both the DataGrid control and the Button control, as the following example shows:

```
function doPrint() {
    var pj : PrintJob = new PrintJob();

    if(pj.start() != true){
        delete pj;
        return;
    }

    pj.addPage(myDataGrid);
    pj.addPage(myButton);
    pj.send();
    delete pj;
}
```

Each call to the `addPage()` method adds a new printed page to the print job. Therefore, the controls print on separate pages.

Modifying a component for printing

A DataGrid control with many rows might not fit on a single screen in your application. In that case, you typically add scroll bars to let users view the entire control when your application executes in Flash Player. By default, when you print the DataGrid control, it prints as it appears on the screen. Therefore, if your DataGrid control has rows or columns that are not visible, they do not print.

You can add logic to your application to modify a component for printing. For example, you can scale a component so that it prints larger than it appears on the screen. If you modify the visual aspects of a component for printing, you should restore the component to its original state so that the user does not see the modifications. For an example that uses scaling, see [“About scaling” on page 614](#).

For a DataGrid control, you can print the control across several pages so that all rows print. The following example uses a `doPrint()` function to print the entire DataGrid control on multiple pages, where each page contains a subset of the rows:

```
function doPrint() {
    var pj : PrintJob = new PrintJob();

    //Save the current vertical scroll position of the DataGrid control.
    var prev_vPosition:Number = myDataGrid.vPosition;

    if(pj.start() != true){
        delete pj;
        return;
    }

    //Calculate the number of visible rows.
    var rowsPerPage:Number = Math.floor((myDataGrid.height -
        myDataGrid.rowHeight)/ myDataGrid.rowHeight);

    //Calculate the number of pages required to print all rows.
```

```

var pages:Number = Math.ceil(myDataGrid.dataProvider.length /
    rowsPerPage);

//Scroll down each page of rows, then call addPage() once for each page.
for (var i=0;i<pages;i++) {
    myDataGrid.vPosition = i*rowsPerPage;
    pj.addPage(myDataGrid);
}

pj.send();
delete pj;

// Restore vertical scroll position.
myDataGrid.vPosition = prev_vPosition;
}

```

This example saves the current vertical scroll position of the DataGrid control, and then resets it just before the function returns. Otherwise, users would see the DataGrid scroll to the bottom of the control every time it prints.

Starting a print job

Calling the `PrintJob.start()` method prompts Flash Player to spool the print job to the user's operating system, and also prompts the user's operating system print dialog box to appear.

If the user selects an option to begin printing from the print dialog box, the `PrintJob.start()` method returns a value of `true`. (The value is `false` if the user cancels the print job, in which case the script should only call `delete`.) If successful, the `PrintJob.start()` method sets values for the `paperHeight`, `paperWidth`, `pageHeight`, `pageWidth`, and `orientation` properties.

Depending on the user's operating system, an additional dialog box might appear until spooling is complete and the `PrintJob.send()` method is called: calls to `PrintJob.addPage()` and then `PrintJob.send()` should be made expeditiously. If 10 seconds elapse between the `PrintJob.start()` method call and the `PrintJob.send()` method call, which sends the print job to the printer, Flash Player effectively calls the `PrintJob.send()` method, which causes any pages that are added using the `PrintJob.addPage()` method to print, and spooling stops.

When a new print job is constructed, the `PrintJob()` method properties are initialized to 0. When the `PrintJob.start()` method is called, after the user selects the print option in the operating system print dialog box, Flash Player retrieves the print settings from the operating system. The `PrintJob.start()` method populates the following properties:

| Property | Type | Unit | Notes |
|-----------------------------------|--------|--------|---|
| <code>PrintJob.paperHeight</code> | Number | Points | Overall paper height. |
| <code>PrintJob.paperWidth</code> | Number | Points | Overall paper width. |
| <code>PrintJob.pageHeight</code> | Number | Points | Height of actual printable area on the page; does not include any user-set margins. |

| Property | Type | Unit | Notes |
|-----------------------------------|--------|--------|--|
| <code>PrintJob.pageWidth</code> | Number | Points | Width of actual printable area on the page; does not include any user-set margins. |
| <code>PrintJob.orientation</code> | String | N/A | Portrait or landscape orientation. |

Note: A *point* is a print unit of measurement that is equal in size to one pixel, a screen unit of measure. For more information about unit equivalencies, see [“About scaling” on page 614](#).

Adding pages to a print job

You add pages to your print job with the `PrintJob.addPage()` method. Although the method can include up to four arguments, **target** is the only required argument. The three optional arguments are **printArea**, **options**, and **frameNum**.

The `addPage()` method uses the following signature:

```
MyPrintJob.addPage(target, printArea:Object, options:Object,
    frameNum:Number):Boolean;
```

Note: Do not set the `frameNum` argument when printing in Flex. Always leave that argument unset, or set it to 0.

Use `NULL` in place of optional arguments that you are not setting. If you provide an invalid argument, the print job uses default argument values, which are specified in the sections that follow.

Each call to add a new page is unique, which lets you modify arguments without affecting previously set arguments. For example, you can specify that one page prints as a bitmap image, and another page prints as a vector graphic. You can add as many new pages to your print job as the print job requires. One call to add a page equals one printed page.

Note: Any `ActionScript` that you have to call to change a resulting printout must run before calling the `PrintJob.addPage()` method. The `ActionScript`, however, can run before or after you create a new `PrintJob` class object.

Specifying a target

The **target** argument can be a number that represents a level (such as 0 for the root document), or a string that represents the instance name of a component.

You can add the entire application to the print job by using the `this` keyword, as the following example shows:

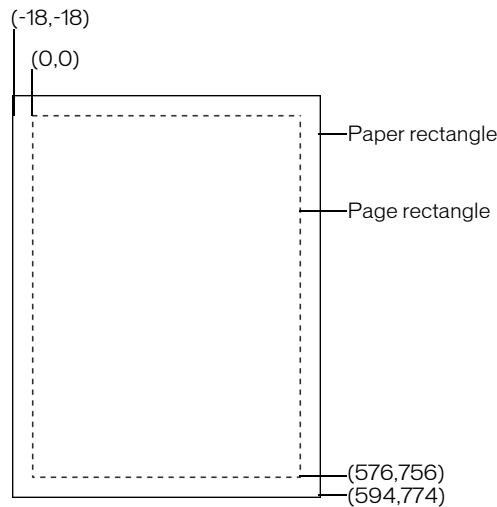
```
pj.addPage(this);
```

Specifying a print area

The **printArea** optional argument includes the following values:

```
{xMin:Number, xMax:Number, yMin:Number, yMax:Number}
```

The `xMin`, `xMax`, `yMin`, and `yMax` values represent screen pixels relative to the target level or upper-left corner of the component. The print area orientation is from the upper-left corner of the printable area on the page. If the print area is larger than the printable area on the page, the print data that exceeds the right and bottom edge of the page is clipped.



If you don't specify a print area, or if you specify an invalid print area, the print area is the Stage area of the root document.

About scaling

A print job that uses the `PrintJob` class prints, by default, without scaling. For example, an object that is 144 pixels wide onscreen prints as 144 points, or 2 inches wide.

To understand how Flash Player screen content maps to the printed page, it helps to understand screen and print units of measure. Pixels are a screen measurement and points are a print measurement. Both pixels and points equal 1/72 of an inch. A *twip* is 1/20 of a point and pixel.

The following list further explains the relationship between these units of measure:

- 1 pixel = 1 point = 20 twips
- 72 pixels = 72 points = 1 inch
- 567 twips = 1 cm
- 1440 twips = 1 inch

To scale a component before printing, set its `scaleX` and `scaleY` properties before you call the `addPage()` method, and then set them back to their original values afterward. The following example scales a `DataGrid` control to 200% before printing:

```
function doPrint() {
    var pj : PrintJob = new PrintJob();

    if(pj.start() != true){
        delete pj;
    }
}
```

```

        return;
    }

    //Save the current scale.
    var currentScaleX:Number = myDataGrid.scaleX;
    var currentScaleY:Number = myDataGrid.scaleY;
    //Scale the DataGrid to 200%.
    myDataGrid.scaleX=200;
    myDataGrid.scaleY=200;

    pj.addPage(myDataGrid);
    pj.send();
    delete pj;

    //Restore the scale.
    myDataGrid.scaleX=currentScaleX;
    myDataGrid.scaleY=currentScaleY;
}

```

If you scale a component and also pass a value for the *printArea* argument, the pixel values passed to *printArea* reflect the original size of the component. That is, if you set a component's scale to 50% and specify a print area of 500 x 500 pixels, the content that prints is identical to the content that would print if you didn't scale the component; however, it prints at half the size.

Specifying printing as a vector image or bitmap graphic

The *options* argument lets you specify whether to print as a vector graphic or bitmap image. When using this optional argument, use the following syntax:

```
{printAsBitmap:boolean}
```

The default value is *false*, which represents a request for vector printing. Keep in mind the following suggestions when determining which value to use:

- If the content that you're printing includes a bitmap image, specify that the print job print as a bitmap to include any transparency and color effects.
- If the content does not include bitmap images, specify that the print job print as vector graphics to take advantage of the higher image quality.

Sending the print job to a printer

To send the print job to a printer after using the *addPage()* method calls, use the *PrintJob.send()* method, which causes Flash Player to stop spooling the print job so that the printer starts printing.

Deleting the print job

After sending the print job to a printer, use the *ActionScript* function *delete PrintJob* to delete the *PrintJob* class object and free memory.

CHAPTER 27

Creating Accessible Applications

You can create Macromedia Flex applications that are accessible to all users, including those with disabilities, using the accessibility features provided with Flex. As you design accessible applications, consider how your users will interact with the content. Visually impaired users, for example, might rely on assistive technology, such as screen readers that provide an audio version of screen content, while hearing-impaired users might read text and captions in the document. Other considerations arise for users with mobility or cognitive impairments.

This chapter describes the accessibility features of Flex.

Contents

| | |
|---|-----|
| Accessibility overview | 617 |
| About screen reader technology | 619 |
| Configuring Flex applications for accessibility | 620 |
| Using accessible components and managers | 621 |
| Creating tab order and reading order | 623 |
| Accessibility for hearing-impaired users | 625 |
| Testing accessible content. | 625 |

Accessibility overview

You create accessible content by using accessibility features included with Flex, by taking advantage of ActionScript designed to implement accessibility, and by following recommended design and development practices. The list of recommended practices that follows is not exhaustive, but suggests common issues to consider. Depending on your audience's needs, additional requirements might arise.

Visually impaired users For visually impaired users, including those with color blindness, keep in mind the following design recommendations:

- Describe the layout of your movie and the individual controls used to navigate through the Macromedia Flash application.
- Design and implement a logical tab order.

- Design the document so that constant changes in content do not unnecessarily cause screen readers to refresh. For example, you should group or hide looping elements.
- Provide captions for narrative audio. Be aware of audio in your document that might interfere with a user being able to listen to the screen reader.
- Ensure that color is not the only means of conveying information. In addition, ensure that foreground and background colors contrast sufficiently to make text readable for people with low vision and color blindness.
- Ensure that controls are device-independent (or accessible by keyboard).

Users with mobility impairment For users with mobility impairment, ensure that controls are device-independent (or accessible by keyboard).

Hearing-impaired users For hearing impaired users, you can caption audio content.

Users with cognitive impairment Users with cognitive impairments vary widely in how they experience difficulty interacting with and understanding content. Many users will benefit from one or more of the following design recommendations:

- Uncluttered design that is easily navigable.
- Graphical imagery that helps convey the purpose and message of the application.
- Multiple methods to accomplish common tasks.

About worldwide accessibility standards

Many countries, including the United States, Australia, Canada, Japan, and countries in the European Union, have adopted accessibility standards based on those developed by the World Wide Web Consortium (W3C). The W3C publishes *Web Content Accessibility Guidelines*, a document that prioritizes actions that designers should take to make web content accessible. For information about the Web Accessibility Initiative, see the W3C website at www.w3.org/WAI.

In the United States, the law that governs accessibility is commonly known as Section 508, which is an amendment to the U.S. Rehabilitation Act. Section 508 prohibits federal agencies from buying, developing, maintaining, or using electronic technology that is not accessible to those with disabilities. In addition to mandating standards, Section 508 lets government employees and the public sue agencies in federal court for noncompliance.

For additional information about Section 508, see the following websites:

- The U.S. government-sponsored website at www.section508.gov
- The Macromedia accessibility web page at www.macromedia.com/macromedia/accessibility/

Viewing the Macromedia Flex Accessibility web page

This chapter contains an introduction to the accessibility features in Flex, and to developing accessible applications. For the latest information on creating and viewing accessible Flex content, including supported platforms, known issues, screen reader compatibility, articles, and accessible examples, see the Macromedia Flash Accessibility web page at www.macromedia.com/macromedia/accessibility/.

About screen reader technology

A screen reader is software designed to navigate through a website and read the web content aloud. Visually impaired users often rely on this technology. You can create content designed for use with screen readers for Windows platforms only. Those viewing your content must have Flash Player 7 or later, and Internet Explorer in Windows 98 or later.

JAWS, from Freedom Scientific, is one example of screen reader software. You can access the JAWS page of the Freedom Scientific website at www.hj.com/fs_products/software_jaws.asp. Another commonly used screen reader program is Window-Eyes, from GW Micro. To access the latest information on Window-Eyes, visit the GW Micro website at www.gwmicro.com.

Note: Flex supports only the JAWS screen reader.

Screen readers help users understand what is contained in a web page or Flex application. Based on the keyboard shortcuts that you define, you can allow users to easily navigate through your application using the screen reader.

Because different screen reader applications use varying methods to translate information into speech, your content will vary in how it's presented to each user. As you design accessible applications, keep in mind that you have no control over how a screen reader behaves. You can only mark up the content in your applications in such a way as to expose the text and ensure that screen reader users can activate the controls. You only have control over the content, not the screen readers. This means that you can decide which objects in the movie are exposed to screen readers, provide descriptions for them, and decide the order in which they are exposed to screen readers. However, you cannot force screen readers to read specific text at specific times or control the manner in which that content is read.

Flash Player and Microsoft Active Accessibility (Windows only)

Flash Player is optimized for Microsoft Active Accessibility (MSAA), which provides a highly descriptive and standardized way for applications and screen readers to communicate. MSAA is available for Windows operating systems only. For more information on Microsoft Accessibility Technology, visit the Microsoft Accessibility website at www.microsoft.com/enable/default.aspx.

The Windows ActiveX (Internet Explorer plug-in) version of Flash Player 7 supports MSAA, but the Windows Netscape and Windows stand-alone players do not.

Flex supports a debug version of Flash Player that can display debugging information during runtime, and generate profiling information so that you can more easily develop applications. However, the debug version of Flash Player does not support accessibility.

Caution: MSAA is currently *not* supported in the opaque windowless and transparent windowless modes. (These modes are options in the HTML Publish Settings panel, available for use with the Windows version of Internet Explorer 4.0 or later, with the Flash ActiveX control.) If you need your Flash content to be accessible to screen readers, avoid using these modes.

Configuring Flex applications for accessibility

This section describes how to enable the accessibility features of Flex and how to configure a screen reader for use with Flex applications.

Enabling accessibility in Flex

By default, Flex accessibility features are not enabled. When you enable accessibility, you do add overhead that can increase the size of the resulting SWF file.

To enable accessibility, you can use one of the following methods:

- Enable accessibility by default for all Flex applications so that all requests return accessible content.

To enable accessibility for all Flex applications, edit the `flex-config.xml` file to set the `<accessible>` property to `true`, as the following example shows:

```
<compiler>
...
<accessible>true</accessible>
...
</compiler>
```

- Enable accessibility on an individual request.

If you have not enabled accessibility for all applications by default, you can enable it on an individual request by setting the `accessible` query parameter to `true`, as the following example shows:

```
http://www.mycompany.com/myflexapp/app1.mxml?accessible=true
```

If you edited the `flex-config.xml` file to enable accessibility by default, you can disable it for an individual request by setting the `accessible` query parameter to `false`, as the following example shows:

```
http://www.mycompany.com/myflexapp/app1.mxml?accessible=false
```

- Enable accessibility when you are using the `mxmmlc` command-line compiler.

When you compile a file using the `mxmmlc` command-line compiler, you can use the `-accessible` option to enable accessibility, as the following example shows:

```
mxmmlc -accessible c:/dev/myapps/mywar.war/app1.mxml
```

For more information on the command-line compiler, see [Chapter 36, “Administering Flex,” on page 795](#).

Configuring a JAWS screen reader for Flex applications

To use a JAWS screen reader with a Flex application, you must download scripts from the Macromedia website before invoking a Flex application. These scripts enable some of the accessibility features of Flex.

You can download these scripts, and the installation instructions, from the Macromedia website at www.macromedia.com/macromedia/accessibility/.

To execute correctly, some of the Flex components require the screen reader to be in Forms mode. These scripts let you switch between Virtual Cursor mode and Forms mode using the Control+Shift+A key sequence.

Using accessible components and managers

To accelerate building accessible applications, Macromedia built support for accessibility into Flex components and managers that automate many of the most common accessibility practices related to labeling, keyboard access, and testing and help ensure a consistent user experience across rich applications. Flex comes with the following set of accessible components and managers:

| Component/ Manager | Notes |
|-----------------------|---|
| Accordion container | Use the Page Up and Page Down keys to move between the individual panes of an Accordion container. When a screen reader encounters an Accordion container, it indicates each pane with the word <i>tab</i> . It indicates the current pane with the word <i>active</i> . When a pane is selected, the user moves to that pane by pressing the Enter key. |
| Button control | Activate the Button control using the Space bar. When using a screen reader, activate the Button controls using the Enter key. |
| CheckBox control | Activate the check box items using the Space bar. When using a screen reader, select the CheckBox control using either the Space bar or the keyboard. |
| ComboBox control | Use the Up and Down Arrow keys to move through the items in the ComboBox drop-down list. To open the ComboBox list, press Control+Down Arrow. When using a screen reader in Forms mode, use the Up and Down Arrow keys to move through the items in the list. To open the ComboBox list, press Control+Down Arrow. |
| ControlBar container | None |
| DataGrid control | Use the arrow keys to highlight the contents, and then move between the individual characters within that field. When using a screen reader, use Control+Shift+A to enter Forms mode. Use the Tab key to move between editable fields in the DataGrid control. To edit a field, use the arrow keys to highlight the contents and then move between the individual characters within that field. |
| DateChooser control | Use Up, Down, Left, and Right Arrow keys to change the selected date. Use Home & End keys to reach the first enabled date in the month and the last enabled date in a month, respectively. Use the Page Up and Page Down keys to reach the previous and next months. When using a screen reader, to move focus to the calendar view, press the Enter key to enter Forms mode. Use Up, Down, Left, and Right Arrow keys to change the selected date. Use Home and End keys to reach the first enabled date in a month and last enabled date in a month, respectively. Use the Page Up and Page Down keys to reach the previous and next months. |

| Component/ Manager | Notes |
|-------------------------------|---|
| DateField control | <p>Use the Space bar to open the DateChooser control and select the appropriate date.</p> <p>With the focus on the DateField control, press Enter to switch to Forms mode.</p> <p>Use the Space bar to open the DateChooser control and select the appropriate date.</p> |
| Form container | None |
| Image control | The image control itself is not accessible, but you can add a ToolTip to it so that the contents of the ToolTip are read. |
| Label control | None |
| Link control | <p>Activate the Link control using the Space bar.</p> <p>When using a screen reader, activate the Link control using the Enter key.</p> |
| List control | <p>Use the Up and Down Arrow keys to move through the items in the menu.</p> <p>When using a screen reader with Forms mode on, use the Up and Down Arrow keys to move through the items in the menu.</p> |
| Menu control | <p>Use the Tab key to bring focus to the Menu control. Next, use the Left or Right Arrow key to select an individual menu. Select submenu items using the Up and Down Arrow keys. To exit from the menu bar, select a submenu item on any menu and press the Tab key.</p> <p>When using a screen reader, use the Tab key to bring focus to the Menu control. Press the Enter key to switch to Forms mode. Use the Left or Right Arrow key to select an individual menu. Select submenu items using the Up and Down Arrow keys. To exit from the menu bar, select a submenu item on any menu and press the Tab key to bring focus back to the menu bar. Press the Tab key again to move to the next item in the tab order.</p> |
| MenuBar control | <p>Use the Tab key to bring focus to the MenuBar control. Next, use the Left or Right Arrow key to select an individual menu. Select submenu items using the Up and Down Arrow keys. To exit from the menu bar, select a submenu item on any menu and press the Tab key.</p> <p>When using a screen reader, use the Tab key to bring focus to the MenuBar control. Press the Enter key to switch to Forms mode. Use the Left or Right Arrow key to select an individual menu. Select submenu items using the Up and Down Arrow keys. To exit from the menu bar, select a submenu item on any menu and press the Tab key to bring focus back to the menu bar. Press the Tab key again to move to the next item in the tab order.</p> |
| NumericStepper control | None |
| Panel container | None |
| RadioButton control | <p>With one radio button selected within a group, press the Enter key to enter that group. Next, use the arrow keys to move between items within that group. The Down and Right Arrow keys move to the next item in a group; the Up and Left Arrow keys will move to a previous item in the group.</p> <p>When using a screen reader, select a radio button by using the Enter key.</p> |

| Component/ Manager | Notes |
|------------------------------|---|
| RadioButton Group control | With one radio button selected within a group, press the Enter key to enter that group. Next, use the arrow keys to move between items within that group. The Down and Right Arrow keys move to the next item in a group; the Up and Left Arrow keys will move to a previous item in the group. When using a screen reader, select a radio button by using the Enter key. |
| TabNavigator container | Use the Page Up and Page Down keys to move between individual panes of the TabNavigator container. When a screen reader encounters a TabNavigator container pane, it indicates each panel with the word <i>tab</i> . It indicates the current pane with the word <i>active</i> . When a pane is selected, the user moves to that pane by pressing the Enter key. |
| Text control | None |
| TextArea control | None |
| TextInput control | None |
| TitleWindow container | None |
| ToolTipManager | When using a screen reader, the contents of a ToolTip are read after the item to which the ToolTip is attached. |
| Tree control | Use the Up and Down Arrow keys to move between items in a Tree control. To open a group, use the Right Arrow key. To close a group, use the Left Arrow key. When using a screen reader, press the Enter key to enter Forms mode. This enables the user to open and close nodes of the Tree control. Use the Up and Down Arrow keys to move between items in a Tree control. To open a group, use the Right Arrow key. To close a group, use the Left Arrow key. |

Creating tab order and reading order

There are two aspects to tab indexing order—the *tab order* in which a user navigates through the web content, and the order in which things are read by the screen reader, called the *reading order*.

Flash Player uses a tab index order from left to right and top to bottom. However, if this is not the order you want to use, you can customize both the tab and reading order using the `tabIndex` property. (In ActionScript, the `tabIndex` property is synonymous with the reading order.)

Tab order You can use the `tabIndex` property of every component to create a tab order that determines the order in which objects receive input focus when a user presses the Tab key.

Reading order You can also use the `tabIndex` property to control the order in which a screen reader reads information about the object (known as the reading order). To create a reading order, you assign a `tabIndex` property to every component in your application. You must create a `tabIndex` property for every accessible object, not just the focusable objects. For example, a Text control must have `tabIndex` properties, even though a user cannot tab to it. If you do not provide a `tabIndex` property for every accessible object, Flash Player ignores all `tabIndex` properties whenever a screen reader is present, and uses the default tab order instead.

Creating accessibility with ActionScript

For accessibility properties that apply to the entire document, you can create or modify a global variable called `_accProps`. For more information on the `_accProps` variable, see *Flex ActionScript Language Reference*.

For properties that apply to a specific object, you can use the syntax `instancename._accProps`. The value of `_accProps` is an object that can include any of the following properties:

| Property | Type | Description |
|---------------------------|---------|---|
| <code>.silent</code> | Boolean | Hides a component from a screen reader when set to <code>true</code> . The default value is <code>false</code> . |
| <code>.forceSimple</code> | Boolean | Hides the children of a component from a screen reader when set to <code>true</code> . The default value is <code>false</code> . |
| <code>.name</code> | String | Specifies a description of the component that is read by the screen reader. When accessible objects do not have a specified name, a screen reader uses a generic word, such as <i>Button</i> . |
| <code>.description</code> | String | Specifies a description for the component that is read by the screen reader. |
| <code>.shortcut</code> | String | Specifies a description of a keyboard shortcut for the component that is read by the screen reader. Entering the description here does not create a keyboard shortcut. To create a keyboard shortcut, see the <code>Key</code> class in <i>Flex ActionScript Language Reference</i> . |

Modifying the `_accProps` variable has no effect by itself. You must also use the `Accessibility.updateProperties()` method to inform screen reader users of Flash Player content changes. Calling the method causes Flash Player to re-examine all accessibility properties, update property descriptions for the screen reader, and, if necessary, send events to the screen reader that indicate changes have occurred.

When updating the accessibility properties of multiple objects at once, you have to include only a single call to the `Accessibility.updateProperties()` method. (Excessive updates to the screen reader can cause some screen readers to become too verbose.)

Implementing screen reader detection with the `Accessibility.isActive()` method

To create content that behaves in a specific way if a screen reader is active, you can use the ActionScript `Accessibility.isActive()` method, which returns a value of `true` if a screen reader is present, and `false` otherwise. You can then design your content to perform in a way that is compatible with screen reader use, such as by hiding child elements from the screen reader.

For more information on the `Accessibility` class, see *Flex ActionScript Language Reference*.

For example, you could use the `Accessibility.isActive()` method to decide whether to include unsolicited animation. *Unsolicited animation* means animation that happens without the screen reader doing anything. This can be very confusing for screen readers.

The `Accessibility.isActive()` method provides asynchronous communication between the Flex application and Flash Player, which means that a slight real-time delay could occur between the time the method is called and the time in which Flash Player becomes active, returning an incorrect value of `false`. To ensure that the method is called correctly, you can do one of the following:

- Instead of using the `Accessibility.isActive()` method when your application first plays, call the method whenever you need to make a decision about accessibility.
- Introduce a short delay of one or two seconds at the beginning of your document to give the application enough time to contact Flash Player.

For example, you can attach this method with an `onFocus` event to a button. This generally gives the SWF file enough time to load and you can safely assume that a screen reader user will tab to the first button or object on the Stage.

Accessibility for hearing-impaired users

To provide accessibility for hearing-impaired users, you can include captions for audio content that is integral to the comprehension of the material presented. A video of a speech, for example, would probably require captions for accessibility, but a quick sound associated with a button probably would not require a caption.

Testing accessible content

When you test your accessible applications, follow these recommendations:

- Ensure that you have enabled accessibility. For more information, see [“Configuring Flex applications for accessibility” on page 620](#).
- If you are creating interactive content, test it and verify that users can navigate your content effectively using only the keyboard. This can be an especially challenging requirement, because different screen readers work in different ways when processing input from the keyboard, which means that your content might not receive keystrokes as you intended. Ensure that you test all keyboard shortcuts.

PART III

Data Access and Interconnectivity

This part describes how to use Macromedia Flex data models and data services.

The following chapters are included:

| | |
|--|-----|
| Chapter 28: Managing Data in Flex | 629 |
| Chapter 29: Binding and Storing Data in Flex | 637 |
| Chapter 30: Using Data Services | 655 |
| Chapter 31: Validating Data in Flex | 703 |
| Chapter 32: Formatting Data. | 725 |

CHAPTER 28

Managing Data in Flex

This chapter introduces Macromedia Flex data management. Data management is a combination of features that provide a powerful way to validate, format, and pass data between Flex applications and external data sources.

Contents

| | |
|---|-----|
| About Flex data management. | 629 |
| Comparing Flex data management to other technologies. | 633 |

About Flex data management

Flex provides the following set of features for working with data in your applications: data services, binding, validation, and formatting. These features let you perform the following tasks using MXML tags:

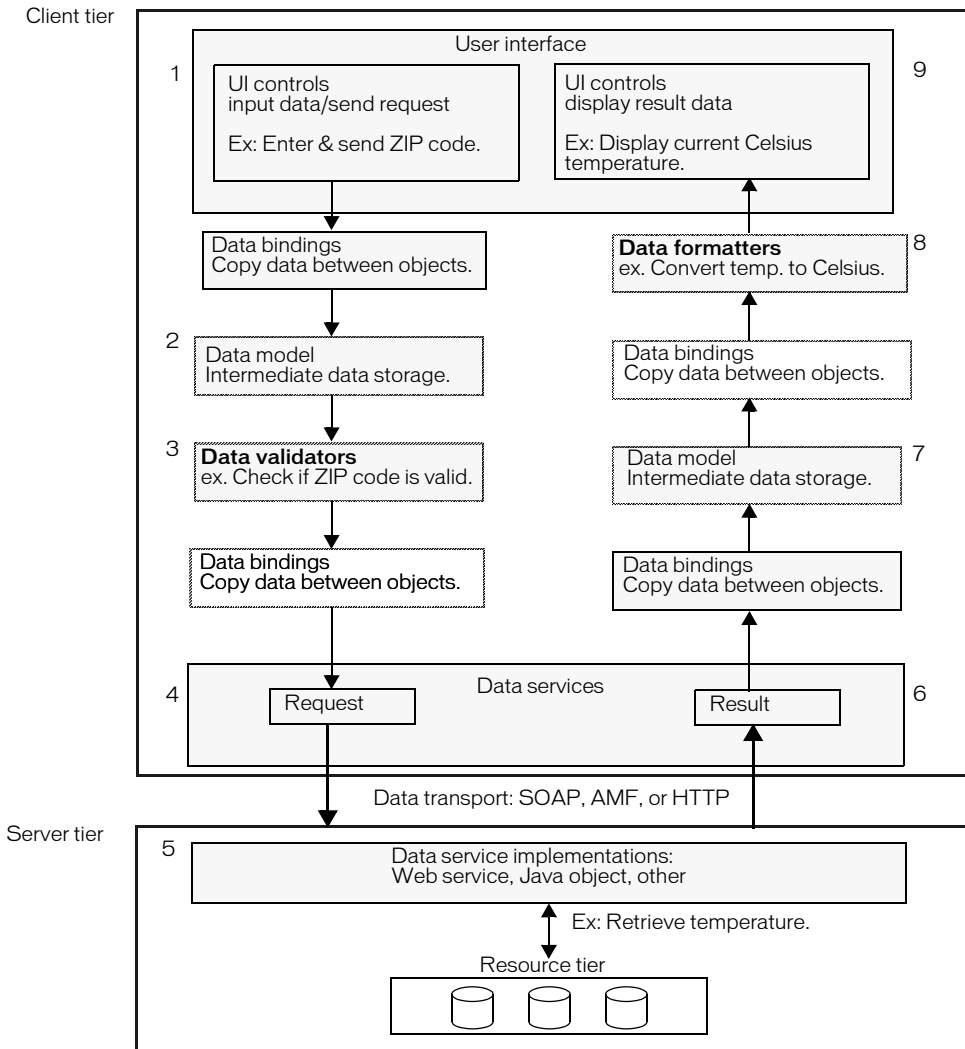
- Send data to server-side data sources
- Receive data from server-side data sources
- Pass data between client-side objects
- Store data in client-side objects
- Validate data before using it
- Format data before displaying it in the user interface

Flex provides tags for connecting to server-side data sources. It also includes a simple syntax for providing input to data source requests and using the data returned from a data source within the client-side components that make up the application.

The following steps describe a simple scenario in which a user provides input data and requests information in a Flex application. A matching figure follows the steps.

1. User enters data in input fields and submits request by clicking a Button control.
2. (Optional) *Data binding* passes data to a data model object, which provides intermediate data storage. This allows data to be manipulated and passed to other objects in the application.
3. (Optional) One or more *data validator* objects validate the request data. Validator objects check whether data meets specific criteria.

4. Data is eventually passed to a *data service* request object.
5. The data service passes the request data to the appropriate method on a server-side object.
6. The server-side object processes the request and returns a result that is converted to a data service result object or a fault object if a valid result cannot be returned.
7. (Optional) Data binding passes data to a data model object, which provides intermediate data storage. This allows data to be manipulated and passed to other objects in the application.
8. (Optional) One or more *data formatter* objects format result data for display in the user interface.
9. Data binding passes data into user interface controls for display.



Data services

The data service feature lets you interact with server-side data sources. You can work with data sources that are accessible using SOAP-compliant web services, Java objects, or HTTP GET or POST requests. Flex is based on a service-oriented architecture (SOA). In a typical Flex application, data is sent as input to one or more external data services. When a data service executes, it returns its results data to the Flex application.

The following example shows MXML code that connects to a data service (a web service in this case), sends a request to the data source in the `click` event of a Button control, and displays the result data in the `text` property of a TextArea control. The value in the curly braces (`{ }`) in the TextArea control is a binding expression that copies data service results data into the `text` property of the TextArea control.

```
...
<!-- Connect to a data service. -->
<mx:WebService id="myService" wsdl="service.wsdl"/>

<!-- Provide input data for calling the web service. -->
<mx:TextInput id="inputText">

<!-- Call the web service, use the text in a TextInput control as input data.
-->
<mx:Button click="myService.getData(inputText.text)">

<!-- Display results data in the user interface. -->
<mx:TextArea text="{myService.getData.result.prop1}"
...

```

For more information, see [Chapter 30, “Using Data Services,” on page 655](#).

Data binding

The data-binding feature provides a syntax for automatically copying the value of a property of one client-side object to a property of another object at runtime. Data binding is usually triggered when the source property value changes. You can use data binding to pass user input data from user interface controls in a Form container to a data service request. You can also use data binding to pass results returned from a data service into user interface controls.

The following example shows a Text control that gets its data from Slider control's `value` property. The property name inside the curly braces (`{ }`) is a binding expression that copies the value of the source property, `mySlider.value`, into the Text control's `text` property.

```
<mx:Slider id="mySlider"/>
<mx:Text text="{mySlider.value}"/>

```

For more information, see [Chapter 29, “Binding and Storing Data in Flex,” on page 637](#).

Data models

The data-model feature lets you store data in client-side objects. A *data model* is an ActionScript object that contains properties for storing data, and optionally contains methods for additional functionality. Data models are useful for partitioning the user interface, data, and data services in an application. You can use the data-binding feature to bind user interface data into a data model; this is particularly useful for validating user input data. You can also use the data-binding feature to bind data service results to a data model.

You can define a simple data model in an MXML tag. When you require functionality beyond storage of untyped data, you can use an ActionScript class as a data model.

The following example shows an MXML-based data model with properties of TextInput controls bound into its fields:

```
...
<mx:Model id="registration">
  <name>{name.text}</name>
  <email>{email.text}</email>
  <phone>{phone.text}</phone>
  <zip>{zip.text}</zip>
  <ssn>{ssn.text}</ssn>
</mx:Model>
...
<mx:TextInput id="name"/>
<mx:TextInput id="email"/>
<mx:TextInput id="phone"/>
<mx:TextInput id="zip"/>
<mx:TextInput id="ssn"/>
...
```

For more information about data models, see [Chapter 29, “Binding and Storing Data in Flex,” on page 637](#). For information about partitioning an application, see Chapter 4, “Architecting Flex Applications” in *Getting Started with Flex*.

Data validation

The data-validation feature lets you ensure that data meets specific criteria before the application uses the data. Data *validators* are ActionScript objects that check whether data is formatted correctly. Validators validate data that is bound to data model fields. For example, you can use a validator to check whether the value that a user enters in a TextInput control is a valid ZIP code before sending it to a data service that expects a ZIP code.

You can apply a data validator to a model declared in a data service declaration, a model declared in an `<mx:Model>` tag, or a model defined in ActionScript. For models in a data service declaration, properties to which a validator component is applied are validated just before the request is sent to a data service, and only valid requests are sent.

The following example shows MXML code that uses the standard `ZipCodeValidator` component, represented by the `<mx:ZipCodeValidator>` tag, to validate the format of the ZIP code that a user enters. The `field` property of the `ZipCodeValidator` validator indicates the property that it validates. When the data binding on the property executes, the validator checks whether the property’s value is a valid ZIP code.


```

...
<mx:TextInput id="input" text="enter zip" width="80"/>

<mx:Model id="zipModel">
    <zip>{input.text}</zip>
</mx:Model>

<mx:ZipCodeValidator field="zipModel.zip" />
...

```

For more information about validator components, see [Chapter 31, “Validating Data in Flex,” on page 703](#).

Data formatting

The data-formatting feature lets you change the format of data before displaying it in a user interface control. For example, when a data service returns a string that you want to display in the (xxx)xxx-xxxx phone number format, you can use a formatter component to ensure that the string is reformatted before it is displayed.

A *data formatter component* is an object that formats raw data into a customized string. You can use data formatter components with data binding to reformat data returned from a data service.

The following example declares a DateFormatter component with an MM/DD/YYYY date format, and binds the formatted version of a Date object returned by a web service to the text property of a TextInput control:

```

...
<!-- Declare a formatter and specify formatting properties. -->
<mx:DateFormatter id="StandardDateFormat" formatString="MM/DD/YYYY" />

<!-- Trigger the formatter while populating a string with data. -->
<mx:TextInput text="Your order shipped on
    {StandardDateFormat.format(myService.purchase.result.date)}" />
</mx:Application>

```

For more information about data formatters, see [Chapter 32, “Formatting Data,” on page 725](#).

Comparing Flex data management to other technologies

The way that Flex works with data sources and data is different from other web application environments, such as JSP, ASP, and ColdFusion. Data access in Flex applications also differs significantly from data access in applications created in Macromedia Flash MX 2004. This section describes some of the differences.

Client-side processing and server-side processing

Unlike a set HTML templates created using JSPs and servlets, ASP, or CFML, the files in a Flex application are compiled on the server into a binary SWF file that is sent to the client. When a Flex application makes a request to an external data service, the SWF file is not recompiled and no page refresh is required.

The following example shows MXML code for calling a web service. When a user clicks the Button control, client-side code calls the web service and result data is returned into the binary SWF file without a page refresh. The result data is then available to use as dynamic content within the application.

```
...
<!-- Define the web service connection (the specified WSDL URL is not
functional). -->
<mx:WebService id="WeatherService" wsdl="/ws/WeatherService?wsdl">
...
<mx:Button label="Get Weather"
    click="WeatherService.GetWeather(input.text);"/>
...
```

The following example shows JSP code for calling a web service using a JSP custom tag. When a user requests this JSP, the web service request is made on the server instead of on the client, and the result is used to generate content in the HTML page. The application server regenerates the entire HTML page before sending it back to the user's web browser.

```
<@ taglib prefix="web" uri="webservicetag" %>

<% String str1="BRL";
String str2="USD";%>

<!-- Call the web service. -->
<web:invoke
    url="http://www.itfinity.net:8008/soap/exrates/default.asp"
    namespace="http://www.itfinity.net/soap/exrates/exrates.xsd"
    operation="GetRate"
    resulttype="double"
    result="myresult">
    <web:param name="fromCurr" value="<%=str1%>"/>
    <web:param name="ToCurr" value="<%=str2%>"/>
</web:invoke>

<!-- Display the web service result. -->
<%= pageContext.getAttribute("myresult") %>
```

Data source access

Another difference between Flex and other web application technologies is that you never communicate directly with a data source in Flex. You use a Flex data service component to connect to a server-side service that interacts with the data source.

The following example shows one way to access a data source directly in a ColdFusion page:

```
...
<CFQUERY DATASOURCE="Dsn"
    NAME="myQuery">
    SELECT * FROM table
</CFQUERY>
...
```

To get similar functionality in Flex, you use an HTTP service, a web service, or a remote object service to call a server-side object that returns results from a data source.

Flash MX data management

Flash MX 2004 and Flex provide different data management architectures. These architectures were developed to meet the needs of the respective authoring environments and user communities. Flash MX 2004 provides a set of data components, which includes XMLConnector, WebServices Connector, DataSet, DataHolder, RDMBSResolver, and XUpdateResolver. These components are designed for use in the Flash MX 2004 authoring environment. Although some of the functionality of these components overlaps with features found in Flex, they are not based on the same architecture.

Flash MX 2004 also has its own data-binding feature that works in conjunction with the Flash MX data components, and is a completely different feature than Flex data binding.

In Flash MX 2004, you can also use Macromedia Flash Remoting MX to connect to server-side data sources. You use the NetServices ActionScript API to work with Flash Remoting MX in Flash MX 2004. When working with the Flex remote object services, you can choose to use the AMF binary transport protocol instead of SOAP. AMF is the protocol used in Flash Remoting MX, but AMF support in remote object services does not include all the features of the Flash Remoting MX product. AMF support in remote object services gives you the choice of a binary protocol when accessing Java objects.

CHAPTER 29

Binding and Storing Data in Flex

This chapter describes the Macromedia Flex data binding and data model features. These features are key components of Flex data management, which let you pass data between objects and store data.

Contents

| | |
|-------------------------|-----|
| Binding data | 637 |
| Using data models | 647 |

Binding data

Data binding is the process of tying the data in one object to another object. It provides a convenient way to pass data around in an application. Flex provides two ways to specify data binding: the curly braces ({}) syntax and the `<mx:Binding>` tag.

Common uses of data binding include the following:

- Properties of user interface controls bound directly to web service requests
- Web service results bound directly to properties of user interface controls
- Web service results bound to a middle-tier data model, and that data model's fields bound to user interface controls; for more information about data models, see [“Using data models” on page 647](#).
- Properties of user interface controls bound to a middle-tier data model, and that data model's fields bound to a web service request (a three-tier system)
- Individual parts of complex properties bound to properties of user interface controls; for example, a master-detail scenario in which clicking an item in a List control displays data in several other controls

Although binding is a powerful mechanism, it is not appropriate for all situations. For example, for a complex user interface in which individual pieces must be updated based on strict timing, you could use a method that assigns properties in order. Also, since binding executes every time a property changes, it is not the best solution when you only want changes to be noticed some of the time.

Data binding requires a source property, a destination property, and a trigger that indicates when to copy the data from the source to the destination. The following example shows a Text control that gets its data from Slider control's `value` property. The property name inside the curly braces (`{ }`) is a binding expression that copies the value of the source property, `mySlider.value`, into the Text control's `text` property.

```
<mx:Slider id="mySlider"/>
<mx:Text text="{mySlider.value}"/>
```

You can bind to all types of properties, including properties of type `Function`. Binding occurs under the following circumstances:

- The object that is the binding source broadcasts an event.
- Application code calls a data service.

Binding data with the curly braces syntax

Using the curly braces syntax is the simplest way to pass data between objects in an application. When using this syntax, you use curly braces (`{ }`) around a source property name as the value of a destination property.

In the following example, a set of properties of user interface controls is bound to the *registration* data model. For more information about data models, see [“Using data models” on page 647](#).

Note: This example is from the Flex Explorer sample application, which is included in the `samples.war` file. You can extract the `samples.war` file to your application server.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">

  <!-- Data model stores registration data that user enters. -->
  <mx:Model id="registration">
    <name>{name.text}</name>
    <email>{email.text}</email>
    <phone>{phone.text}</phone>
    <zip>{zip.text}</zip>
    <ssn>{ssn.text}</ssn>
  </mx:Model>

  <!-- Form contains user input controls. -->
  <mx:Form>
    <mx:FormItem label="Name" required="true">
      <mx:TextInput id="name" width="200"/>
    </mx:FormItem>

    <mx:FormItem label="Email" required="true">
      <mx:TextInput id="email" width="200"/>
    </mx:FormItem>

    <mx:FormItem label="Phone" required="true">
      <mx:TextInput id="phone" width="200"/>
    </mx:FormItem>

    <mx:FormItem label="Zip" required="true">
```

```

        <mx:TextInput id="zip" width="60"/>
    </mx:FormItem>

    <mx:FormItem label="Social Security" required="true">
        <mx:TextInput id="ssn" width="200"/>
    </mx:FormItem>

    <mx:FormItem>

<!-- User clicks Button to place order. -->
        <mx:Button label="Place Order"
            click="mx.validators.Validator.
                isStructureValid(this,'registration');"/>
    </mx:FormItem>
</mx:Form>
</mx:Application>

```

Flex supports ActionScript expressions in bindings. Binding expressions in curly braces can contain an ActionScript expression that returns a value. For example, you can use the curly braces syntax for the following types of binding:

- A single bindable property inside curly braces
- String concatenation that includes a bindable property inside curly braces
- Calculations on a bindable property inside curly braces
- Conditional operations that evaluate a bindable property value

The following example shows a data model that uses each type of binding expression:

```

<mx:Model id="myModel">
    <!-- Perform simple property binding. -->
    <a>{name.text}</a>
    <!-- Perform string concatenation. -->
    <b>This is {name.text}</b>
    <!-- Perform a calculation. -->
    <c>{baz * 6 / 7}</c>
    <!-- Perform a conditional operation using a ternary operator;
        the person object contains a Boolean variable called isMale. -->
    <d>{(person.isMale) ? "Mr." : "Ms."} {person.lastName}</d>
</mx:Model>

```

Binding data with the <mx:binding> tag

You can use the <mx:Binding> tag as an alternative to the curly braces syntax. When you use the <mx:Binding> tag, you provide a source property in the <mx:Binding> tag's source property and a destination property in its destination property. This is equivalent to using the curly braces syntax.

Unlike the curly braces syntax, you can use the <mx:Binding> tag to completely separate the view (user interface) from the model. The <mx:Binding> tag also lets you bind different source properties to the same destination property.

In the following example, the properties of user interface controls are bound to the myEmployee data model using <mx:Binding> tags:

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://macromedia.com/2003/mxml">
...
<!-- Form contains user input controls. -->
  <mx:Form label="Employee Information">
    <mx:FormItem label="First Name">
      <mx:TextInput id="firstName" />
    </mx:FormItem>
    <mx:FormItem label="Last Name">
      <mx:TextInput id="lastName" />
    </mx:FormItem>
    <mx:FormItem label="Department">
      <mx:TextInput id="department" />
    </mx:FormItem>
    <mx:FormItem label="Email Address">
      <mx:TextInput id="email" />
    </mx:FormItem>
  </mx:Form>

  <!-- The myEmployee data model. -->
  <mx:Model id="myEmployee">
    <name>
      <first />
      <last />
    </name>
    <department />
    <email />
  </mx:Model>
...
  <!-- Properties of user interface controls are bound to the
  myEmployee data model using <mx:Binding> tags. -->

  <mx:Binding source="firstName.text" destination="myEmployee.name.first" />
  <mx:Binding source="lastName.text" destination="myEmployee.name.last" />
  <mx:Binding source="department.text" destination="myEmployee.department" />
  <mx:Binding source="email.text" destination="myEmployee.email" />
</mx:Application>

```

Binding more than one source property to a destination property

You can bind more than one source property to the same destination property by using multiple `<mx:Binding>` tags that specify the same destination but different sources, or by using the combination of binding expressions in curly braces and `<mx:Binding>` tags. You cannot do this with just the curly braces syntax.

In the following example, the data model field `thing1.part` is the destination, and both `input1.text` and `input2.text` are its sources. If `input1.text` or `input2.text` is updated, `thing1.part` contains the updated value.

```

<mx:Model id="thing1">
  <part>{input1.text}</part>
</mx:Model>

<mx:Binding source="input2.text" destination="thing1.part" />

```


Using ActionScript expressions in Binding tags

The `source` property of an `<mx:Binding>` tag can contain curly braces. When there are no curly braces in the `source` property, the value is treated as a single ActionScript expression. When there are curly braces in the `source` property, the value is treated as a concatenated ActionScript expression. The `<mx:Binding>` tags in the following example are valid and equivalent to each other:

```
<mx:Binding source="The dog ate my " + dog.whatDogAte()"
            destination="field1.text" />
```

```
<mx:Binding source="{The dog ate my " + dog.whatDogAte()}"
            destination="field1.text" />
```

```
<mx:Binding source="The dog ate my {dog.whatDogAte()}"
            destination="field1.text" />
```

The `source` property in the following example is not valid because it is not an ActionScript expression:

```
<mx:Binding source="The dog ate my homework" destination="field1.text" />
```

About the binding mechanism

Binding expressions must be on fields of typed objects. At compile time, ActionScript Watcher and Binding objects are declared. At runtime, Watcher objects trigger Binding objects to execute bindings. Binding works best with typed variables. For an untyped variable, such as `var foo;`, the binding mechanism cannot detect change events on properties, and it raises a warning. When a field is typed as `Object`, such as `var foo:Object;`, the binding mechanism makes assumptions; for example, it assumes that any property on `foo` is not a getter/setter property. If the property is a getter/setter, unexpected results can occur.

To ensure the best possible binding results, you should always strongly type your variables. When you use one of the standard Flex classes, such as any of the `List` or `DataProvider` classes, you should know that the `selectedItem` property is typed as `Object`. If your `selectedItem` is a real custom class, not a model or data service result, you should cast it, as the binding expression in the following example shows:

```
{MyClass(myList.selectedItem).someProp}
```

Working with bindable property chains

When you specify a property as the source of a data binding, Flex monitors not only that property for changes, but also the chain of properties leading up to it. The entire chain of properties, including the destination property, is called a *bindable property chain*. In the following example, `firstName.text` is a bindable property chain that includes both `firstName` component and its `text` property:

```
<first>{firstName.text}</first>
```

You should raise an event when any named property in a bindable property chain changes. If the property is a normal object property, the Flex compiler generates the event for you. If the property uses a getter/setter pair, you must specify `[ChangeEvent]` metadata that indicates what event is raised when the property changes. Otherwise, the compiler issues a warning and does not trigger a binding for that property. The following example shows how to specify `[ChangeEvent]` metadata:

```
[ChangeEvent("textChanged")]
public function get text() : String
{
    return myText;
}

public function set text(t : String) : Void
{
    myText = t;
    dispatchEvent({type: "textChanged"});
}
```

You can also provide the compiler with better information about an object by casting the object to a known type. In the following example, the `myList` List control contains Customer objects, so the `selectedItem` property is cast to a Customer object:

```
<mx:Model id="selectedCustomer">
    <name>{Customer(myList.selectedItem).name}</name>
    <address>{Customer(myList.selectedItem).address}</address>
    ...
</mx:Model>
```

There are some situations in which binding does not execute automatically as expected. Binding does not execute automatically when you change an entire item of a `dataProvider` property, as the following example shows:

```
dataProvider[i] = newItem
```

Binding also does not execute automatically when you are binding data to a property that Macromedia Flash Player updates automatically, such as the `mouseX` property.

The `executeBindings()` method of the `UIObject` and `Repeater` classes executes all the bindings into a `UIObject` component or `Repeater` object. All containers and controls extend the `UIObject` class. The `executeChildBindings()` method of the `Container` and `Repeater` classes executes all of the bindings into all the child `UIObject` components of a `Container` or `Repeater` class. All containers extend the `Container` class. These methods give you a way to execute bindings that do not occur as expected. By adding one line of code, such as a call to `executeChildBindings()` method, you can update the user interface after making a change that does not cause bindings to execute. However, you should only use the `executeBindings()` method when you are sure that bindings will not execute automatically.

Binding data to and from Arrays

You can bind data to and from Arrays. For example, you can have a data model that contains a repeated element (an Array) and bind that element to another object that accepts Arrays. You can also bind data from an Array into a single element of a data model.

The following example shows the two-way nature of Array binding. A repeated data model element, `<third>`, is bound to the `dataProvider` property of a List control; this displays the content of that Array in the List control.

The content of the List control is then bound to an element called `<array>` in another data model; this stores the Array in that single data model element. The `<array>` element is then bound to the `dataProvider` property of a second List control. This displays the echoed Array in the second List control.

```
<?xml version="1.0"?>
  <mx:Application width='700' height='600'
    xmlns:mx="http://www.macromedia.com/2003/mxml" >
    <mx:Model id="mod">
      <first>Label 1</first>
      <second>Label 2</second>
      <third>Label 3.1</third>
      <third>Label 3.2</third>
      <third>{t1.text}</third>
    </mx:Model>

    <mx:Model id="mod2">
      <array>{list1.dataProvider}</array>
    </mx:Model>

    <mx:TextInput id="t1" text="Label 3.3"/>
    <mx:List id="list1">
      <mx:dataProvider>{mod.third}</mx:dataProvider>
    </mx:List>

    <mx:List id="list2">
      <mx:dataProvider>{mod2.array}</mx:dataProvider>
    </mx:List>
  </mx:Application>
```

You can also store an Array in an `ActionScript` variable, and then bind that variable into a data model element, as the following example shows:

```
<?xml version="1.0"?>
<mx:Application width='700' height='600'
xmlns:mx="http://www.macromedia.com/2003/mxml" >

  <mx:Script>
    <![CDATA[
      var someArray = ["hi", "there"];
    ]]>
  </mx:Script>
  <mx:Model id="mod">
    <array>{someArray}</array>
  </mx:Model>

</mx:Application>
```

Note: Array elements do not trigger ChangeEvents and, therefore, cannot function as binding sources at runtime. Binding copies initial values during instantiation after variables are declared in an `<mx:Script>` tag, but before handlers are executed. Arrays that are bound do not stay updated if individual fields of a source Array change.

Using binding to pass data between objects

Many applications require a messaging strategy for moving related data between objects. You can use binding with messaging objects, whose sole purpose is to move related data around an application.

For example, suppose you are building a library card catalog system, and you want to let users search a library database. The first thing you do is provide the users with a search form. At the same time, you create a simple Query object in an ActionScript class. The Query object has fields for author, title, and subject. Because the Query object only holds typed data, it does not need a lot of additional functionality.

The user is only going to create one query at a time, so you declare the Query object as a custom ActionScript component inside an MXML component. You then bind the search form elements into the properties of the Query object, as the following example shows:

```
<?xml version="1.0"?>
<!-- QueryForm.mxml -->

<mx:VBox xmlns:mx="http://www.macromedia.com/2003/mxml" xmlns:library="*">
  <library:Query id="myQuery">
    <library:author>{authorInput.text}</library:author>
    <library:title>{titleInput.text}</library:title>
    <library:subject>{subjectInput.text}</library:subject>
  </library:Query>

  <mx:Form>
    <mx:FormItem label="Author">
      <mx:TextInput id="authorInput" />
    </mx:FormItem>
    ...
  </mx:Form>
  <mx:Button label="Submit Query"
    click="queryBizDelegate.sendQuery(myQuery)" />
</mx:VBox>
```

You could also include validators and any other user interface logic that you require in this form. The main idea is that instead of having to know the parameters of the `sendQuery()` method, you can pass a Query object, which can be filled in using binding, to the form.

Going in the other direction, suppose the library query service returns a result that contains a collection of books. Some separate logic can navigate to the results page, and that page might have a DataGrid control to display all of the books that are returned. The service can simply drop off the books at a known location. It does not need to know that there is a view that cares about the result. The following example shows the service result handler:

```
function queryResult(event) : Void
{
    queryBizDelegate.acceptResult(event.result);
}
```

The `queryBizDelegate.acceptResult()` method stores the result in a well-known location, such as a `queryResults` property. You can bind the `queryResults` property to a `DataGrid` control's `dataProvider` property so that the view automatically updates whenever new results arrive, as the following example shows:

```
<mx:DataGrid dataProvider="{queryBizDelegate.queryResults}" />
```

Considerations

Consider the following when using the binding feature:

- When bindings contain errors, the Flex compiler emits identical warnings about unknown properties multiple times. This happens because the compiler places the binding code in two functions, so errors show up in two contexts. One of those contexts is removed later during the compilation process.
- Because the `SharedObject` object is built into Flash Player, its behavior cannot be modified by the binding mechanism. To use binding with a `SharedObject`, you must write a wrapper object that uses a `SharedObject` object internally, but has getter/setter properties that the rest of your application can use. If you do not use a wrapper object, a `SharedObject` appears to lose data, specifically data that might have been used in a binding expression.
- Validators and effects are not accessible by an `id` property, and binding is not allowed on them.
- You must use casting when binding to an object in which a variable is typed as `Object` if the object has a getter/setter property. This affects `Repeater` objects and other objects with a `dataProvider` property. For more information about casting, see [“Working with bindable property chains” on page 641](#).
- The `selectedItem` property of the `List` class and the `currentItem` property of `Repeater` class are typed explicitly as `Object`. Binding works correctly when the property is a simple object with name-value pairs. However, binding fails if the property is a typed object in which a bound property is a getter/setter property. Because the variable is explicitly typed to `Object`, no warnings are raised. When binding to the `selectedItem` or `currentItem` property, and the property you are binding to is implemented as a getter/setter, cast to the appropriate class, as the following example shows:

```
{MyGetterClass(theList.selectedItem).myGetterProp}
```

- When you use a binding expression with a string concatenation, binding does not execute when the string concatenation contains the word *undefined*. If you want the word *undefined* to appear in the text, you must use something like `"{'undefined is some text' + myValue}"` instead of `"undefined is some text {myValue}"`.
- If a component uses binding expressions, the bindings execute before the component's event executes, and destinations are updated before their component's event is executed. However, you could bind a component to an object that you haven't created yet. If you must rely on all bindings having run before your code fragment executes, you can use the root tag's event.

- The Flex compiler cannot detect dynamic properties created using the `Object.addProperty()` method. Using these properties with bindings may cause incorrect behavior. Consider whether you can use getter/setter functionality along with `[ChangeEvent]` metadata instead. For more information, see [“Working with bindable property chains” on page 641](#).
- You cannot bind to styles.
- If you bind a `Model` into the `dataProvider` property of a component, you should not change items in the `Model` directly. Instead, change the items through the `DataProvider` API. Otherwise, the component to which the `Model` is bound is not redrawn to show the changes to the `Model`. For example, instead of using `myGrid.getItemAt(itemIndex).myField = 1;`, you would use `myGrid.dataProvider.editField(itemIndex, "myField", 1);`.

Debugging data binding

In some situations, data binding might not appear to function correctly. The following list contains suggestions for resolving data binding issues:

- Pay attention to warnings.
It is easy to see a warning and think that it doesn't matter, especially if binding appears to work at startup, but warnings are important.
If a warning is about a missing `[ChangeEvent]` on a getter/setter property, even if the binding works at startup, subsequent changes to the property are not noticed.
If a warning is about a static variable or a built-in property, changes aren't noticed.
If you are sure that the property should be bindable but the compiler is complaining about an unknown type, you might need to cast an object in your binding expression so that the compiler can find the appropriate type information, including `[ChangeEvent]` metadata. For more information about casting, see [“Working with bindable property chains” on page 641](#).
- Ensure that the source of the binding actually changed.
When your source is part of a larger procedure, it is easy to miss the fact that you never assigned the source.
- Ensure that the `ChangeEvent` event is being dispatched.
You can use the Flex debugger, `fdb`, or another debugger to make sure that the `dispatchEvent()` method is called. Also, you can add a normal event listener to that class to make sure it gets called. If you want to add the event listener as a tag attribute, you must place the `[Event('myEvent')]` metadata at the top of your class definition or in an `<mx:Metadata>` tag in your MXML.
- Create a setter function and use a `<mx:Binding>` tag to assign into it.
You can then put a trace or an alert or some other debugging code in the setter with the value that is being assigned. This technique ensures that the binding itself is working. If the setter is called with the right information, you now know that it's your destination that is failing and you can start debugging in there.

Using data models

A Flex *data model* is an ActionScript object that contains properties that you use to store application-specific data. You can use a data model for data validation, and it can contain client-side business logic. You can define a data model in MXML or ActionScript. In the model-view-controller (MVC) design pattern, the data model represents the model tier.

When you plan an application, you determine the kinds of data that the application must store and how that data must be manipulated. This helps you decide what types of data models you need. For example, suppose you decide that your application must store data about employees. A simple employee model might contain name, department, and e-mail address properties.

Defining a data model

You can define a data model in an MXML tag, ActionScript function, or an ActionScript class. In general, you should use MXML-based models for simple data structures and use ActionScript for more complex structures and client-side business logic.

Note: The `<mx:Model>` and `<mx:XML>` tags are Flex compiler tags and do not correspond directly to ActionScript classes. The *Flex ActionScript and MXML API Reference* documentation, included in the documentation.zip file, contains information about these tags and other compiler tags; click MXML Tags at the top of the main *Flex ActionScript and MXML API Reference* page.

You can place an `<mx:Model>` tag or an `<mx:XML>` tag in a Flex application file or in an MXML component file. The tag should have an `id` value, and it cannot be the root tag of an MXML component.

Model tag

The most common type of MXML-based model is the `<mx:Model>` tag, which is compiled into an object that contains a tree of ActionScript objects. The leaves of the tree are scalar values. The following example shows an employee model declared in an `<mx:Model>` tag:

```
<mx:Model id="employee">
  <name>
    <first />
    <last />
  </name>
  <department />
  <email />
</mx:Model>
```

An `<mx:Model>` child tag with no value is considered null. If you want an empty string instead, you can use a binding expression as the value of the tag, as the following example shows:

```
<mx:Model id="employee">
  <name>
    <!--Fill the first property with empty string.-->
    <first>{""}</first>
    <!--Fill the last property with empty string.-->
    <last>{""}</last>
  </name>
  <!--department is null-->
  <department />
```

```

        <!--email is null-->
        <email />
    </mx:Model>

```

If you supply one instance of a child tag in an `<mx:Model>` tag, there is no way for Flex to know if you intend it to be an array or a single property instance. You can work around this limitation by using an `<mx:Object>` tag instead of an `<mx:Model>` tag and declaring an `<mx:Array>` tag inside the `<mx:Object>` tag, as the following example shows. The `<mx:Object>` tag and `<mx:Array>` tag declare standard ActionScript Object and Array objects, respectively. You could also use an ActionScript class to work around the same limitation of the `<mx:Model>` tag.

```

<mx:Object id="model1">
    <employees>
        <mx:Array>
            <mx:Object>
                <name>
                    <mx:Object>
                        <first>
                            <mx:String></mx:String>
                        </first>
                        <last>
                            <mx:String></mx:String>
                        </last>
                    </mx:Object>
                </name>
                <department>
                    <mx:String></mx:String>
                </department>
                <email>
                    <mx:String></mx:String>
                </email>
            </mx:Object>
        </mx:Array>
    </employees>
</mx:Object>

```

XML tag

An `<mx:XML>` tag represents literal XML data in an ActionScript `XMLNode` object. It is currently easier to declare a model in an `<mx:Model>` tag and manipulate it as a tree of ActionScript objects instead of an `XMLNode` object. For more information about the `XMLNode` class, see the *Flex ActionScript Language Reference*.

Script-based models

As an alternative to using an MXML-based model, you can define a model as a variable in an `<mx:Script>` tag. The following example shows a very simple model defined in an ActionScript script block. It would be easier to declare this model in an `<mx:Model>` tag.

```

<mx:Script>
    var myEmployee={
        name:{
            first:undefined,
            last:undefined

```



```

    },
    department:undefined,
    email:undefined
  };
</mx:Script>

```

Class-based models

Using an ActionScript class as a model is a good option when you want to store complex data structures or you want to execute client-side logic using application data. The following example shows a model defined in an ActionScript class. This model is used to store shopping cart items in an e-commerce application. It also provides methods for adding and removing items, getting an item count, and getting the total price. For more information on ActionScript components, see [Chapter 14, “Creating ActionScript Components,” on page 359](#).

Note: This example is from the Flex Store application included in the samples.war file. You can extract the samples.war file to your application server.

```

class ShoppingCart {

    var items : Array;
    var total : Number = 0;

    function ShoppingCart() {
        items=new Array();
    }

    function addItem(item : Object, qty : Number, index: Number) : Void {
        qty=qty==null?1:qty;
        index=index==null?0:index;
        items.addItemAt(index, {id: item.id, name: item.name,
            description: item.description, image: item.image, price: item.price,
            qty: qty==null?1:qty});
        total+=parseFloat(item.price)*qty;
    }

    function removeItemAt(index) {
        total-=parseFloat(items[index].price)*items[index].qty;
        items.removeItemAt(index);
    }

    function getItemCount() : Number {
        return items.length;
    }

    function getTotal() : Number {
        return total;
    }

}

```

You declare a class-based model as an ActionScript component tag in an MXML file, as the following example shows:

```
<local:ShoppingCart id="cart" xmlns:local="*" />
```

This component that is in the same directory as the MXML file, as indicated by the XML namespace value `*`. For more information about specifying the location of components, see [Chapter 14, “Creating ActionScript Components,” on page 359](#).

Specifying an external source

You specify an external source for a data model in a `source` property. The external source can contain static data and data binding expressions, just like a model defined in the body of the `<mx:Model>` or `<mx:XML>` tag. The file referenced in a `source` property resides on the server and not on the client machine. The compiler reads the `source` value and compiles the source into the application; the `source` value is not read at runtime. To retrieve XML data at runtime, you can use the `<mx:HTTPService>` tag; for more information, see [Chapter 30, “Using Data Services,” on page 655](#).

Using `<mx:Model>` and `<mx:XML>` tags with external sources is an easy way to reuse data model structures and data binding expressions. You can also use them to prepopulate user interface controls with static data by binding data from the model elements into the user interface controls.

The `source` property accepts the names of files relative to the current web application directory, as well as URLs with `HTTP://` and `file://` prefixes. In the following example, the content of the `myEmployee1` data model is an XML file named `content.xml` in the local web application directory. The content of the `myEmployee2` data model is a fictional HTTP URL that returns XML:

```
<mx:Model source="employees.xml" id="employee1" />
```

```
<mx:Model source="http://www.somesitel.com/employees.xml" id="employee2" />
```

The source file must be a valid XML document with a single root node. Flex renders everything inside the first node, so it is equivalent to the contents being in the tag *without* the first node. The following example shows an XML file that could be used as the source of the `<mx:Model source="employees.xml" id="Model1" />` tag. The root node, `<employees>`, is ignored and its contents are copied into the data model.

Note: This example is from the Flex Explorer sample application included in the `samples.war` file. You can extract the `samples.war` file to your application server.

```
<?xml version="1.0"?>
<employees>

  <employee>
    <name>Christina Coenraets</name>
    <phone>555-219-2270</phone>
    <email>ccoenraets@fictitious.com</email>
    <active>true</active>
  </employee>
  <employee>
    <name>Louis Freligh</name>
```

```

        <phone>555-219-2100</phone>
        <email>lfreligh@fictitious.com</email>
        <active>true</active>
    </employee>
    <employee>
        <name>Ronnie Hodgman</name>
        <phone>555-219-2030</phone>
        <email>rhodgman@fictitious.com</email>
        <active>false</active>
    </employee>
    <employee>
        <name>Joanne Wall</name>
        <phone>555-219-2012</phone>
        <email>jwall@fictitious.com</email>
        <active>true</active>
    </employee>
    <employee>
        <name>Maurice Smith</name>
        <phone>555-219-2012</phone>
        <email>maurice@fictitious.com</email>
        <active>false</active>
    </employee>
    <employee>
        <name>Mary Jones</name>
        <phone>555-219-2000</phone>
        <email>mjones@fictitious.com</email>
        <active>true</active>
    </employee>
</employees>

```

Using validators with a data model

To validate the data stored in a data model, you use validators. Binding data from user input controls into a data model provides an easy way to validate the data. For more information about validators, see [Chapter 31, “Validating Data in Flex,” on page 703](#).

In the following example, the `<mx:EmailValidator>`, `<mx:PhoneNumberValidator>`, `<mx:ZipCodeValidator>`, and `<mx:SocialSecurityValidator>` tags declare validators that validate the *email*, *phone*, *zip*, and *ssn* fields of the *registration* data model. The validators generate error messages when a user enters incorrect data in `TextInput` controls that are bound to the data model fields.

Note: This example is from the Flex Explorer sample application included in the `samples.war` file. You can extract the `samples.war` file to your application server.

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">

    <!-- Data model stores registration data that user enters. -->
    <mx:Model id="registration">
        <name>{name.text}</name>
        <email>{email.text}</email>
        <phone>{phone.text}</phone>
    </mx:Model>

```

```

        <zip>{zip.text}</zip>
        <ssn>{ssn.text}</ssn>
    </mx:Model>

    <mx:EmailValidator field="registration.email"/>
    <mx:PhoneNumberValidator field="registration.phone"/>
    <mx:ZipCodeValidator field="registration.zip"/>
    <mx:SocialSecurityValidator field="registration.ssn"/>

    <!-- Form contains user input controls. -->
    <mx:Form>
        <mx:FormItem label="Name" required="true">
            <mx:TextInput id="name" width="200"/>
        </mx:FormItem>

        <mx:FormItem label="Email" required="true">
            <mx:TextInput id="email" width="200"/>
        </mx:FormItem>

        <mx:FormItem label="Phone" required="true">
            <mx:TextInput id="phone" width="200"/>
        </mx:FormItem>

        <mx:FormItem label="Zip" required="true">
            <mx:TextInput id="zip" width="60"/>
        </mx:FormItem>

        <mx:FormItem label="Social Security" required="true">
            <mx:TextInput id="ssn" width="200"/>
        </mx:FormItem>

        <mx:FormItem>
            <!-- User clicks Button to place order. -->
            <mx:Button label="Place Order"
                click="mx.validators.Validator.
                    isStructureValid(this,'registration');" />
        </mx:FormItem>
    </mx:Form>
</mx:Application>

```

This example cleanly separates the user interface and application-specific data. You could easily extend it to create a three-tier architecture by bind data from the registration data model into a data service request. You could also bind user input data directly into a data service request, which itself is a data model, as described in [Chapter 30, “Using Data Services,” on page 655](#).

Using a data model as a value object

You can use a data model as a value object, which acts as a central repository for a set of data returned from method calls on one or more objects. This makes it easier for you to manage data and move it around an application.

Note: The following examples are from the Data Model application included in the samples.war file. You can extract the samples.war file to your application server.

In the following example, the *tentModel* data model stores the results of a web service operation. The *TentDetail* component is a custom MXML component that gets its data from the *tentModel* data model and displays details for the currently selected tent.

```
...
<!-- Data model stores data from selected tent. -->
<mx:Model id="tentModel">
    <name>{selectedTent.name}</name>
    <sku>{selectedTent.sku}</sku>
    <capacity>{selectedTent.capacity}</capacity>
    <season>{selectedTent.seasonStr}</season>
    <type>{selectedTent.typeStr}</type>
    <floorarea>{selectedTent.floorArea}</floorarea>
    <waterproof>{getWaterProof(selectedTent.waterProof)}</waterproof>
    <weight>{getWeight(selectedTent)}</weight>
    <price>{selectedTent.price}</price>
</mx:Model>
...
<TentDetail id="detail" tent="{tentModel}"/>
...
```

The following example shows the MXML source code for the *TentDetail* component. References to the *tent* property, which contains the *tentModel* data model, and the corresponding *tentModel* properties are highlighted.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Panel xmlns:mx="http://www.macromedia.com/2003/mxml" title="Tent Details">

    <mx:Script>
        var tent:Object;
    </mx:Script>

    <mx:Style>
        .title{fontFamily:Arial;fontWeight:bold;color:#3D3D3D;fontSize:16pt;}
        .flabelColor
            {fontFamily:Arial;fontWeight:bold;color:#3D3D3D;fontSize:11pt}
        .productSpec{fontFamily:Arial;color:#5B5B5B;fontSize:10pt}
    </mx:Style>

    <mx:VBox marginLeft="10" marginTop="10" marginRight="10">

        <mx:Form verticalGap="0" marginLeft="10" marginTop="10"
            marginRight="10" marginBottom="0">

            <mx:VBox width="209" height="213">
                <mx:Image width="207" height="211"
                    source="./images/{tent.sku}_detail.jpg"/>
            </mx:VBox>

            <mx:FormHeading label="{tent.name}" verticalGap="1"
                styleName="title"/>

            <mx:HRule width="209"/>

            <mx:FormItem label="Capacity" styleName="flabelColor">
```

```

        <mx:Label text="{tent.capacity}" person"
        styleName="productSpec"/>
    </mx:FormItem>
    <mx:FormItem label="Season"
        styleName="flabelColor">
        <mx:Label text="{tent.season}"
            styleName="productSpec"/>
    </mx:FormItem>
    <mx:FormItem label="Type" styleName="flabelColor">
        <mx:Label text="{tent.type}"
            styleName="productSpec"/>
    </mx:FormItem>
    <mx:FormItem label="Floor Area" styleName="flabelColor">
        <mx:Label text="{tent.floorarea}
            square feet" styleName="productSpec"/>
    </mx:FormItem>
    <mx:FormItem label="Weather" styleName="flabelColor">
        <mx:Label text="{tent.waterproof}"
            styleName="productSpec"/>
    </mx:FormItem>
    <mx:FormItem label="Weight" styleName="flabelColor">
        <mx:Label text="{tent.weight}"
            styleName="productSpec"/>
    </mx:FormItem>
</mx:Form>
</mx:VBox>
</mx:Panel>

```

Binding data into an XML document

Flex compiles the `<mx:XML>` tag into literal XML data in an ActionScript XMLNode object. This is different from the `<mx:Model>` tag, which Flex compiles into an Action object that contains a tree of ActionScript objects. To bind data into an `<mx:XML>` data model, you can use the curly braces syntax the same way you do with other data models. However, you cannot use a node within the data model as a binding source. Macromedia does not recommend using the `<mx:Binding>` tag for this type of binding because that requires you to write an appropriate ActionScript XML command as the destination property of the `<mx:Binding>` tag. For more information about the `<mx:XML>` tag, see [“Defining a data model” on page 647](#).

The following example shows an `<mx:XML>` data model with binding destinations in curly braces:

```

...
<mx:XML id="myEmployee">
    <name>
        <first>{firstName.text}</first>
        <last>{lastName.text}</last>
    </name>
    <department>{department.text}</department>
    <email>{email.text}</email>
</mx:XML>
...

```

Note: You cannot bind one piece of XML into another.

CHAPTER 30

Using Data Services

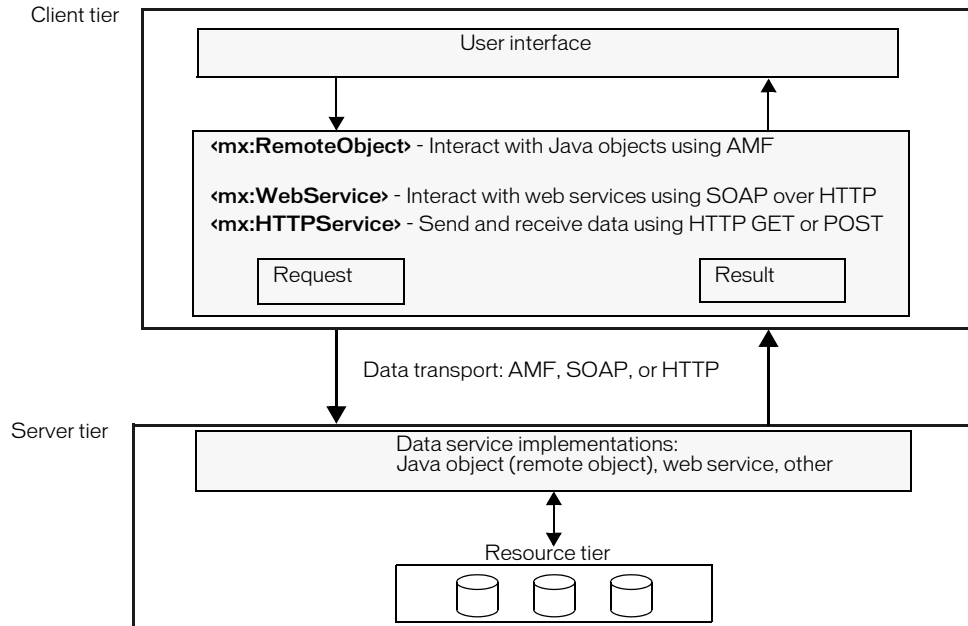
Macromedia Flex is based on a service-oriented architecture in which you use data services to interact with server-side data sources. You can work with data sources that are accessible using Java objects, Simple Object Access Protocol (SOAP)-compliant web services, or Hypertext Transfer Protocol (HTTP) GET or POST requests. In a typical Flex application, client-side data service objects send requests to external data services, which return result data to the client-side objects. This chapter describes the data service features of Flex.

Contents

| | |
|--|-----|
| About data services | 656 |
| Declaring a data service | 658 |
| Calling a data service | 659 |
| Handling data service results | 665 |
| Using a service with binding, validation, and event handlers | 669 |
| Handling asynchronous calls to data services | 670 |
| Using callback URLs | 672 |
| Generating debugging information for data services | 673 |
| Working with remote object services | 674 |
| Working with web services | 681 |
| Securing data services | 687 |
| Data service tag properties | 696 |
| Data service whitelist tags | 701 |

About data services

You can use MXML tags to work with three types of server-side data services: remote object services, web services, and HTTP services. This section briefly describes the different types of data services and when to use them. The following figure provides a simplified view of how Flex uses data services to interact with server-side data sources:



The following list describes some of the key considerations when you are creating an application that must access server-side data:

1. What is the best type of data service to use? For more information, see [“Remote object services” on page 657](#), [“Web services” on page 657](#), and [“HTTP services” on page 657](#).
2. Do you want to use a named or unnamed service? For more information, see [“Declaring a data service” on page 658](#).
3. What is the best way to pass data to the service? For more information, see [“Calling a data service” on page 659](#).
4. How do you want to handle data results from the service? For more information, see [“Handling data service results” on page 665](#).
5. How can you debug your data services code? For more information, see [“Generating debugging information for data services” on page 673](#).
6. What security measures should you implement? For more information, see [“Securing data services” on page 687](#).

Remote object services

Remote object services let you access the methods of server-side Java objects, such as plain old Java objects (POJOs) and JavaBeans, without manually configuring the objects as web services. You can use a remote object service instead of a web service when objects are not already published as web services, web services are not used in your environment, or you would rather use Java objects than web services. You can use the `<mx:RemoteObject>` tag to connect to a local Java object that is in the Flex web application's classpath.

You can use the `<mx:RemoteObject>` tag to interact with Java objects using the Action Message Format (AMF) encoding. When you use the `<mx:RemoteObject>` tag, Flex sends messages in a binary format.

For more specific information about using remote object services, see [“Working with remote object services” on page 674](#).

Web services

Web services are software modules with methods, commonly referred to as *operations*; their interfaces are defined using XML. Web services provide a standards-compliant way for software modules running on a variety of platforms to interact with each other. For more information about web services, see the web services section of the World Wide Web Consortium's website at www.w3.org/2002/ws/.

Flex applications can interact with web services that define their interfaces in a Web Services Description Language (WSDL) document, which is available as a URL. WSDL is a standard format for describing the messages that a web service understands, the format of its responses to those messages, the protocols that the web service supports, and where to send messages.

Flex applications support web service requests and results that are formatted as SOAP messages and are transported over HTTP. SOAP provides the definition of the XML-based format that you can use for exchanging structured and typed information between a web service client, such as a Flex application, and a web service.

You can use the `<mx:WebService>` tag to connect to a SOAP-compliant web service when web services is an established standard in your environment. The `<mx:WebService>` tag is also useful for objects that are within an enterprise environment, but not necessarily available to the Flex web application's classpath.

For more specific information about using web services, see [“Working with web services” on page 681](#).

HTTP services

HTTP services let you send HTTP GET and POST requests, and include data from HTTP responses in a Flex application.

HTTP services are a good option when you cannot expose the same functionality as a SOAP web service or a remote object service. For example, you can use HTTP services to interact with JavaServer Pages (JSPs), servlets, and ASP pages that are not available as web services or remote object services.

You can use the `<mx:HTTPService>` tag for CGI-like interaction in which you use HTTP GET or POST to send a request to a specified URL. When you call the `HTTPService` object's `send()` method, it makes an HTTP GET or POST request to the specified URL, and an HTTP response is returned. Optionally, you can pass arguments to the specified URL.

Declaring a data service

You declare a data service in a data service tag. Flex applications access data services as unnamed or named services. You define an unnamed service entirely in the `<mx:RemoteObject>`, `<mx:WebService>`, or `<mx:HTTPService>` tag in an MXML application. To authenticate users, you must use a named service. A named service relies on explicit server configuration. For descriptions of data service tag properties, see [“Data service tag properties” on page 696](#).

Unnamed services

You define an unnamed service entirely in an `<mx:RemoteObject>`, `<mx:WebService>`, or `<mx:HTTPService>` tag.

The following example shows an unnamed remote object service declaration. The `source` property specifies a fully qualified Java class name that is in the classpath.

```
<mx:RemoteObject id="MyService" source="credit.CreditCardAuth"/>
```

The following example shows an unnamed web service declaration. It specifies the service's identifier and WSDL document in the `<mx:WebService>` tag:

```
<mx:WebService id="MyService" wsdl="http://somewhere.com/my.wsdl"/>
```

You can use a relative or absolute URL in the `wsdl` property to access an unnamed web service. You can specify a relative URL in the following ways:

- Just the WSDL filename specifies a location relative to the directory that contains the application MXML file
- The WSDL filename preceded by a forward slash (/) specifies a location relative to the web root of the server from which the Flex application is served
- A URL that begins with `@ContextRoot()/` specifies a location relative to the context root of the web application from which the Flex application is served

The following example shows an unnamed HTTP service declaration. The value of the `url` property can be a relative or absolute URL. A relative URL is relative to the context root of the web application and must begin with `@ContextRoot()`.

```
<mx:HTTPService id="MyService" url="@ContextRoot()/directory/myfile.xml"/>
```

Named services

Named service declarations for remote object services use the `named` property instead of the `source` property. Named service declarations for web services and HTTP services use the `serviceName` property instead of the `wsdl` or `url` property.

The following example shows a named remote object service declaration:

```
<mx:RemoteObject id="employeeR0" named="SalaryR0"
    result="empList=event.result">
    <mx:method name="getList"/>
</mx:RemoteObject>
```

You define named services in `<whitelist>` tags in the `flex-config.xml` file. The `<remote-objects>`, `<web-service-proxy>`, and `<http-service-proxy>` tags in the `flex-config.xml` file each contain a `<whitelist>` tag that contains a `<named>` tag. For remote object services, the `<named>` tag contains child `<object>` tags for each named service. For web services and HTTP services, the `<named>` tag contains child `<service>` tags for each named service.

The following example sets up a remote object service named `SalaryR0`:

```
<remote-objects>
...
  <whitelist>
    ...
    <named>
      <object name="SalaryR0">
        <source>samples.explorer.SalaryManager</source>
        <type>stateful-class</type>
      </object>
    </named>
  </whitelist>
...
</remote-objects>
```

For more information about whitelists for each type of data service, see [“Data service whitelist tags” on page 701](#).

Calling a data service

Regardless of the source of input data, calling a data service requires an ActionScript function, called an event handler, which makes a data service request when an ActionScript event occurs. The two general categories of events are user events and system events. *User events* occur as a result of user interaction with the application; for example, a click on a Button control is a user event. *System events* occur as a result of systematic code execution.

Flex provides two ways to call a data service: *explicit argument passing* and *argument binding*. When you use explicit argument passing, you provide input to service operations or methods in the form of arguments to an ActionScript function. This is a common way to work with remote object services because it closely resembles the way you call methods in Java. You can also use explicit argument passing with web services and HTTP services. Unlike web services and remote object services, when you use explicit argument passing with HTTP services, you specify arguments in a `send()` method. You cannot use the Flex data validation feature in combination with explicit argument passing.

When you use argument binding, you declare remote object service methods, web service operations, or HTTP service request arguments in `<mx:method>` tags, `<mx:operation>` tags, or `<mx:request>` tags, respectively. You call a `send()` method to send the request.

You can use remote object service `<mx:method>` tags or web service `<mx:operation>` tags with either explicit argument passing or argument binding to set the properties described in the following table. The `name` property is the only property that is required.

| Property | Description |
|--------------------------|---|
| <code>concurrency</code> | <p>Value that indicates how to handle multiple calls to the same method. By default, making a new request to an operation or method that is already executing does not cancel the existing request.</p> <p>The following values are permitted:</p> <ul style="list-style-type: none"> • <code>multiple</code> Existing requests are not cancelled, and the developer is responsible for ensuring the consistency of returned data by carefully managing the event stream. This is the default value. • <code>single</code> Making only one request at a time is allowed on the method; multiple requests generate a fault. • <code>last</code> Making a request cancels any existing request. <p>The request referred to here is not the HTTP request. It is the client action request, or the <code>pendingCall</code> object. HTTP requests are sent to the server and get processed on the server side. However, the result is ignored when a request is cancelled; requests are cancelled when you use the <code>single</code> or <code>last</code> value described above. The <code>last</code> request is not necessarily the last one that the server receives over HTTP.</p> |
| <code>fault</code> | ActionScript code that runs when an error occurs. |
| <code>name</code> | (Required) The name of the operation or method to call. |
| <code>result</code> | ActionScript code that runs when a result object is available. The result object is passed in as an event argument. |

Using explicit argument passing

When you use explicit argument passing, you provide input to a service in the form of arguments to an ActionScript function. The following example passes the data of a selected ComboBox item to the `employeeR0.getList()` method when the user clicks a Button control:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    verticalGap="10">
    <mx:Script>
        <![CDATA[
            var empList;
        ]]>
    </mx:Script>
    <mx:RemoteObject id="employeeR0"
        source="samples.explorer.EmployeeManager" result="empList=event.result"
        fault="alert(event.fault.faultstring, 'Error')">
    </mx:RemoteObject>
    <mx:HBox>
        <mx:Label text="Select a department:"/>
        <mx:ComboBox id="dept" width="150">
            <mx:dataProvider>
                <mx:Array>
                    <mx:Object label="Engineering" data="ENG"/>
                    <mx:Object label="Product Management" data="PM"/>
                    <mx:Object label="Marketing" data="MKT"/>
                </mx:Array>
            </mx:dataProvider>
        </mx:ComboBox>
    </mx:HBox>
</mx:Application>
```

```

        </mx:Array>
    </mx:dataProvider>
</mx:ComboBox>
<mx:Button label="Get Employee List"
    click="employeeRO.getList(dept.selectedItem.data)"/>
</mx:HBox>
<mx:DataGrid dataProvider="{empList}" width="100%">
    <mx:columns>
        <mx:Array>
            <mx:DataGridColumn columnName="name" headerText="Name"/>
            <mx:DataGridColumn columnName="phone" headerText="Phone"/>
            <mx:DataGridColumn columnName="email" headerText="Email"/>
        </mx:Array>
    </mx:columns>
</mx:DataGrid>
</mx:Application>

```

Note: The Flex Samples Explorer application included in the samples.war file contains this example. You can extract the samples.war file to your application server.

Explicit argument passing with remote object services and web services

An `<mx:RemoteObject>` tag or `<mx:WebService>` tag can contain `<mx:method>` tags or `<mx:operation>` tags, respectively. In these tags, you can specify concurrency, fault, and result properties. You can also specify default values for concurrency, fault, and result in the `<mx:RemoteObject>` tag or `<mx:WebService>` tag and use no child tags at all.

Explicit argument passing with HTTP services

You use an `HTTPService` object's `send()` method to call an HTTP service with either explicit argument passing or argument binding. When you use explicit argument passing, you can specify an object that contains name-value pairs as a `send()` method argument. A `send()` method argument must be a simple base type; you cannot use complex nested objects because there is no generic way to convert them to name-value pairs.

If you do not specify an argument to the `send()` method, the `HTTPService` object uses any query arguments specified in an `<mx:request>` tag.

The following examples show two ways to call an HTTP service using the `send()` method with an argument. The second example also shows how to call the `cancel()` method to cancel HTTP service calls.

```

<!-- HTTP service call with an object as a send() method argument that provides
    query arguments. -->
<mx:Button click="MyService.send({param1: 'val1'})";

<!-- HTTP service call with a send() method that takes a variable as its
    argument. The value of the variable is an Object. -->
<mx:Script>
<![CDATA[
    myFunction(){
        var params = new Object();
        params.param1 = 'val1';
        MyService.send(params);
    }

```

```

        // Cancel all existing service calls.
        MyService.cancel();
    }

]]>
</mx:Script>

```

Using argument binding

When you use argument binding, you provide request arguments in child tags of an `<mx:RemoteObject>`, `<mx:WebService>`, or `<mx:HTTPService>` tag. Argument binding lets you copy data from user interface controls or models to request arguments. Because request argument tags represent a data model, you can apply validators to argument values before submitting requests to data services. When you apply a validator, Flex validates the request just before sending it, and only sends valid requests. For more information about data binding and data models, see [Chapter 29, “Binding and Storing Data in Flex,” on page 637](#). For more information about data validation, see [Chapter 31, “Validating Data in Flex,” on page 703](#).

Argument binding with remote object services and web services

When you use argument binding with remote object services or web services, you always declare methods or operations in a remote object service `<mx:method>` tag or a web service `<mx:operation>` tag.

An `<mx:method>` tag can contain an `<mx:arguments>` tag that contains child tags for the method arguments. An `<mx:operation>` tag can contain an `<mx:request>` tag that contains the XML nodes that the operation expects. The name property of an `<mx:method>` tag or an `<mx:operation>` tag must match one of the remote object service method names or one of the web service operation names, respectively.

For remote object services, the order of the argument tags must match the order of the Java method arguments. You can name argument tags to match the actual names of the corresponding Java method arguments as closely as possible, but this is not necessary.

If argument tags inside an `<mx:arguments>` tag have the same name, service calls fail if the remote method is not expecting an Array as the only input source. There is no warning about this when the application is compiled.

You can bind data to remote object service method arguments or web service operation arguments. You use the tag names of the arguments for data binding and validation. The following example shows a remote object service method with two arguments bound to the text properties of TextInput controls. A `PhoneNumberValidator` validator is assigned to `arg1`, which is the name of the first argument tag.

```

...
<mx:RemoteObject id="ro"...>
    <mx:method name="setData">
        <mx:arguments>
            <arg1>{text1.text}</arg1>
            <arg2>{text2.text}</arg2>
        </mx:arguments>
    </mx:method>
</mx:RemoteObject>

```

```

    </mx:method>
<mx:RemoteObject>
...
<mx:PhoneNumberValidator field="ro.setData.arg1"/>
...

```

Flex sends the remote object service argument values to the service method in the order specified in the MXML tags.

The following example uses argument binding in a web service `<mx:operation>` tag to bind the data of a selected ComboBox item to the `employeeWS.getList` operation when the user clicks a Button control. When you use argument binding, you call a service by using the `send()` method with no arguments. For a remote object service, you use an `<mx:method>` tag and `<mx:arguments>` tag in place of the `<mx:operation>` tag and `<mx:request>` tag.

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    verticalGap="10">
    <mx:WebService id="employeeWS"
        wsdl="@ContextRoot()/services/EmployeeWS?wsdl"
        showBusyCursor="true"
        fault="alert(event.fault.faultstring)">
        <mx:operation name="getList">
            <mx:request>
                <deptId>{dept.selectedItem.data}</deptId>
            </mx:request>
        </mx:operation>
    </mx:WebService>
    <mx:HBox>
        <mx:Label text="Select a department:"/>
        <mx:ComboBox id="dept" width="150">
            <mx:dataProvider>
                <mx:Array>
                    <mx:Object label="Engineering" data="ENG"/>
                    <mx:Object label="Product Management" data="PM"/>
                    <mx:Object label="Marketing" data="MKT"/>
                </mx:Array>
            </mx:dataProvider>
        </mx:ComboBox>
        <mx:Button label="Get Employee List" click="employeeWS.getList.send()"/>
    </mx:HBox>
    <mx:DataGrid dataProvider="{employeeWS.getList.result}" width="100%">
        <mx:columns>
            <mx:Array>
                <mx:DataGridColumn columnName="name" headerText="Name"/>
                <mx:DataGridColumn columnName="phone" headerText="Phone"/>
                <mx:DataGridColumn columnName="email" headerText="Email"/>
            </mx:Array>
        </mx:columns>
    </mx:DataGrid>
</mx:Application>

```

Note: The Flex Samples Explorer application included in the samples.war file contains this example. You can extract the samples.war file to your application server.

Argument binding with HTTP services

When a service takes query arguments, you can declare them as child tags of an `<mx:request>` tag. The names of the tags must match the names of the query arguments that the service expects.

The following example uses argument binding in the `<mx:request>` tag of an `<mx:HTTPService>` tag to bind the data of a selected `ComboBox` item to the `employeeSrv` request when the user clicks a `Button` control. When you use argument binding, you call a service using the `send()` method with no arguments.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    verticalGap="20">
    <mx:HTTPService id="employeeSrv" url="employees.jsp">
        <mx:request>
            <deptId>{dept.selectedItem.data}</deptId>
        </mx:request>
    </mx:HTTPService>
    <mx:HBox>
        <mx:Label text="Select a department:"/>
        <mx:ComboBox id="dept" width="150">
            <mx:dataProvider>
                <mx:Array>
                    <mx:Object label="Engineering" data="ENG"/>
                    <mx:Object label="Product Management" data="PM"/>
                    <mx:Object label="Marketing" data="MKT"/>
                </mx:Array>
            </mx:dataProvider>
        </mx:ComboBox>
        <mx:Button label="Get Employee List" click="employeeSrv.send();"/>
    </mx:HBox>
    <mx:DataGrid dataProvider="{employeeSrv.result.employees.employee}"
        width="100%">
        <mx:columns>
            <mx:Array>
                <mx:DataGridColumn columnName="name" headerText="Name"/>
                <mx:DataGridColumn columnName="phone" headerText="Phone"/>
                <mx:DataGridColumn columnName="email" headerText="Email"/>
            </mx:Array>
        </mx:columns>
    </mx:DataGrid>
</mx:Application>
```

Note: The Flex Samples Explorer application included in the `samples.war` file contains this example. You can extract the `samples.war` file to your application server.

Using the Flex proxy for web services and HTTP services

By default, web service requests and HTTP service requests go through the Flex proxy. The *proxy* provides functionality that lets you access URLs on different domains and manage security. Remote object services do not use the proxy. For named services, the proxy generates a URL and appends the service name to the end, as the following example shows:

`http://your_machine_name/flex/flashproxy/MyService`

When you do not require the functionality provided by the proxy, you may want to bypass it. You can bypass the proxy by setting the `useProxy` property to `false` in an `<mx:WebService>` tag or `<mx:HTTPService>` tag, depending on the value of the corresponding `<proxy-use-policy>` tag in the `<web-service-proxy>` tag or `<http-service-proxy>` tag in the `flex-config.xml` file. The `<proxy-use-policy>` tag accepts the following values:

| Value | Description |
|---------------------|--|
| <code>client</code> | (Default) The value of the <code>useProxy</code> property determines if the proxy is used. If you do not specify the <code>useProxy</code> property in the <code><mx:WebService></code> tag or <code><mx:HTTPService></code> tag, Flex uses the proxy. |
| <code>always</code> | Flex always uses the proxy. When you set the <code>useProxy</code> property to <code>false</code> in the <code><mx:WebService></code> tag or <code><mx:HTTPService></code> tag, Flex generates a warning. |
| <code>never</code> | Flex never uses the proxy. When you set the <code>useProxy</code> property to <code>true</code> in the <code><mx:WebService></code> tag or <code><mx:HTTPService></code> tag, Flex generates a warning. |

Note: You must use the Flex proxy to correctly return HTTP status codes from a service.

Handling data service results

After a service operation executes, the data that the service returns is placed in a result object. By default, the data returned is represented as a simple tree of `ActionScript` objects. Flex interprets the XML data that a web service or HTTP service returns to appropriately represent base types, such as `String`, `Number`, `Boolean`, and `Date`. To work with more complex objects, you must populate those objects using the object tree that Flex creates.

For web services, much of the total round-trip time for a complex service call comes from decoding the returned XML into `ActionScript` objects, and many complex results involve large data sets, so each element is only decoded when it is requested. For example, in a situation where a `DataGrid` control displays 10 items of a 1000 item recordset, only the first 10 items of the XML are decoded into `ActionScript`. This reduces the total perceived time to draw the initial `DataGrid` to almost nothing. There is no perceived performance gain if the whole data set must be decoded for an operation such as sorting the data set.

Note: Because ColdFusion is case insensitive, it internally uppercases all of its data. Keep this in mind when consuming a ColdFusion web service.

Binding a service result object to other objects

You can bind properties of the service result object to the properties of other objects, including user-interface components and data models. Whenever a service request executes, the result object is updated and any associated bindings are also updated.

In the following example, two properties of the result object, `CityShortName` and `CurrentTemp`, are bound to the `text` properties of two `TextArea` controls. The `CityShortName` and `CurrentTemp` properties are returned when a user makes a request to the `MyService.GetWeather()` operation and provides a ZIP code as an operation request argument.

```
<mx:TextArea text="{MyService.GetWeather.result.CityShortName}"/>
<mx:TextArea text="{MyService.GetWeather.result.CurrentTemp}"/>
```

You can bind an Array contained in a service result object to a complex property of a user interface component, such as a List, ComboBox, or DataGrid control. In the following example, an Array of objects, `employeeWS.getList.result`, is bound to the `dataProvider` property of a DataGrid control to display employee names, phone numbers, and e-mail addresses in the DataGrid control.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    verticalGap="10">
    <mx:WebService id="employeeWS" wsdl="@ContextRoot()/services
        EmployeeWS?wsdl"
        showBusyCursor="true"
        fault="alert(event.fault.faultstring)">
        <mx:operation name="getList">
            <mx:request>
                <deptId>{dept.selectedItem.data}</deptId>
            </mx:request>
        </mx:operation>
    </mx:WebService>
    <mx:HBox>
        <mx:Label text="Select a department:"/>
        <mx:ComboBox id="dept" width="150">
            <mx:dataProvider>
                <mx:Array>
                    <mx:Object label="Engineering" data="ENG"/>
                    <mx:Object label="Product Management" data="PM"/>
                    <mx:Object label="Marketing" data="MKT"/>
                </mx:Array>
            </mx:dataProvider>
        </mx:ComboBox>
        <mx:Button label="Get Employee List" click="employeeWS.getList.send()"/>
    </mx:HBox>
    <mx>DataGrid dataProvider="{employeeWS.getList.result}" width="100%">
        <mx:columns>
            <mx:Array>
                <mx>DataGridColumn columnName="name" headerText="Name"/>
                <mx>DataGridColumn columnName="phone" headerText="Phone"/>
                <mx>DataGridColumn columnName="email" headerText="Email"/>
            </mx:Array>
        </mx:columns>
    </mx>DataGrid>
</mx:Application>
```

If you are unsure whether the result of a data service contains an Array or an individual object, you can use the `toArray()` method of the `mx.utils.ArrayUtil` class to convert it to an Array. If you pass the `toArray()` method to an individual object, it returns an Array with that object as the only Array element. If you pass it an Array, it returns the same Array.

The following example uses the `toArray()` method as part of the binding statement:

```
<mx:ComboBox>
    <mx:dataProvider>{mx.utils.ArrayUtil.toArray(myResults)}</mx:dataProvider>
</mx:ComboBox>
```

You usually do not need to use the `toArray()` method for web services because there is a schema associated with the web service that specifies to Flex whether an item is an Array, even if it contains only a single object.

Binding a complex result object to a data model

To create a clean separation between data services and the user interface, you can bind a service result object to a data model, and then bind that data model to user-interface components. This gives you the opportunity to filter or massage data before displaying it, or use data for multiple purposes.

The following code sample is from an application that displays a tent image for each tent object that a web service returns. When the user clicks a tent image, its properties are bound to the *tentModel* data model. Using data binding, the *TentDetail* control gets data from the *tentModel* data model, and displays details for the currently selected tent.

Note: The Data Model application included in the samples.war file contains this example. You can extract the samples.war file to your application server.

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
  xmlns="*" pageTitle="Tents">
  ...
  <mx:Model id="tentModel">
    <name>{selectedTent.name}</name>
    <sku>{selectedTent.sku}</sku>
    <capacity>{selectedTent.capacity}</capacity>
    <season>{selectedTent.seasonStr}</season>
    <type>{selectedTent.typeStr}</type>
    <floorarea>{selectedTent.floorArea}</floorarea>
    <waterproof>{getWaterProof(selectedTent.waterProof)}</waterproof>
    <weight>{getWeight(selectedTent)}</weight>
    <price>{selectedTent.price}</price>
  </mx:Model>
  <mx:Array id="allTentsArray">{ws.getAllTents.result}</mx:Array>
  <mx:VBox width="0%">
    <mx:Label text="Tents" styleName="appTitle"/>
    <mx:HBox>
      <mx:Panel title="Tent Collection" id="allTentsPanel" width="570"
        height="490">
        <mx:Canvas>
          <mx:Tile id="tentTile" marginLeft="20" marginTop="10"
            verticalGap="10" horizontalGap="20" marginRight="55"
            marginBottom="90">
            <mx:Repeater id="allTents" count="20"
              dataProvider="{allTentsArray}">
              <TentComp id="tc"
                index="{allTents.currentIndex}"
                imageName="{allTents.currentItem.sku}"
                mouseOver="showHover(event.target.index)"
                mouseDown="setDetail(event.target.index)"/>
            </mx:Repeater>
          </mx:Tile>
          <TentComp id="dark" visible="false">
```

```

        mouseOver="showHover(event.target.index)"/>
        <TentHover id="hover" visible="false"
        mouseDown="setDetail(event.target.index)"
        mouseOut="hideHover();" tentTile="{tentTile}"/>
    </mx:Canvas>
</mx:Panel>
<TentDetail id="detail" tent="{tentModel}"
    height='{allTentsPanel.height}'/>
</mx:HBox>
...
</mx:Application>

```

Handling request-level events

When a service call is completed, the `RemoteObject`, `WebService`, or `HTTPService` object raises a *result event* or a *fault event*. A *result event* indicates that the result is available. A *fault event* indicates that an error occurred. The *result event* acts as a trigger to update properties that are bound to the result object. You can handle *fault* and *result* events explicitly by attaching functions to a remote object service `<mx:method>` tag or web service `<mx:operation>` tag. For an HTTP service, you specify *fault* and *result* event handlers on the `<mx:HTTPService>` tag itself because an HTTP service cannot have multiple operations or methods.

When no event handler is specified for *result* or *fault* events at the request level, the events are passed to the top level of the service; for web services and remote object services, you can specify service-level *result* and *fault* event handlers in the `<mx:RemoteObject>` tag or `<mx:WebService>` tag, respectively. When no event handlers are specified at the service level, Flex dispatches an error event with a message that the Application object can handle; by default, the Application object displays an error message in an Alert pop-up dialog box.

In the following example, the *result* and *fault* properties of a web service `<mx:operation>` tag specify event handlers:

```

<mx:WebService id="WeatherService" wsdl="/ws/WeatherService?wsdl">
    <mx:operation name="GetWeather"
        fault="showErrorDialog(event.fault.faultstring)"
        result="log(event.result)">
        <mx:request>
            <ZipCode>{myZipField.text}</ZipCode>
        </mx:request>
    </mx:operation>
</mx:WebService>

```

Alternatively, you can specify event handlers for *result* and *fault* events in an `<mx:Script>` tag. You can also place a result handler at the individual call level (see [“Handling asynchronous calls to data services” on page 670](#)).

For information about scoping in `ActionScript`, see [Chapter 12, “Working with ActionScript in Flex,” on page 319](#).

Using a service with binding, validation, and event handlers

You can validate data before passing it to a data service, and broadcast an event when the data service returns a result or a fault. The following example shows an application that validates service request data and assigns an event handler to `result` and `fault` events.

This two-tier application does the following:

1. Declares a web service.
2. Binds user interface data to a web service request.
3. Validates a ZIP code.
4. Binds data from a web service result to a user interface control.
5. Specifies `result` and `fault` event handlers for a web service operation.

You can also create three-tier applications that use an additional data-model layer between the user interface and the web service. For more information about data models and data binding, see [Chapter 28, “Managing Data in Flex,” on page 629](#).

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
    width="600" height="400">

    <!-- Web service object handles web service requests and results
         (the specified WSDL URL is not functional). -->
    <mx:WebService id="WeatherService" wsdl="/ws/WeatherService?wsdl">
        <mx:operation name="GetWeather"
            fault="showErrorDialog(event.faultstring)" result="log();">

    <!-- The mx:request data model stores web service request data. -->
        <mx:request>
            <ZipCode>{myZipField.text}</ZipCode>
        </mx:request>
    </mx:operation>
</mx:WebService>

    <!-- Validator validates ZIP code using the standard mx:ZipCode validator. -->
    <mx:ZipCodeValidator field="WeatherService.GetWeather.request.ZipCode"/>
    <mx:VBox>
    <mx:TextInput id="myZipField" text="enter zip" width="80"/>

    <!-- Button triggers web service request. -->
    <mx:Button id="mybutton" label="Get Weather"
        click="WeatherService.getWeather.send();"/>

    <!-- TextArea control displays the results the web service returns. -->
    <mx:TextArea id="temp" text="The current temperature in
        {WeatherService.GetWeather.result.CityShortName} is
        {WeatherService.GetWeather.result.CurrentTemp}."
        height="30" width="200"/>
    </mx:VBox>
</mx:Script>
```

```

<![CDATA[
    function log() {
        // function implementation
    }

    function showErrorDialog(error){
        // function implementation
    }
]]>
</mx:Script>

</mx:Application>

```

Handling asynchronous calls to data services

Because ActionScript code executes asynchronously, if you allow concurrent calls to a data service, you must ensure that your code handles the results appropriately based on the context in which the service is called. By default, making a request to a web service operation that is already executing does not cancel the existing request. In a Flex application in which a service can be called from multiple locations, the service might respond differently in different contexts.

When you design a Flex application, consider whether the application requires disparate data sources and the number of types of services that the application requires. The answers to these questions help determine the level of abstraction that you provide in the data layer of the application.

In a very simple application, user-interface components might call data services directly. In applications that are slightly larger, business objects might call data services. In still larger applications, business objects might interact with service broker objects that call data services.

To understand the results of asynchronous service calls to objects in an application, you need a good understanding of scoping in ActionScript. For more information, see [Chapter 12, “Working with ActionScript in Flex,” on page 319](#).

Using the Asynchronous Completion Token design pattern

Because Flex is a service-oriented framework in which code executes asynchronously, it lends itself well to the Asynchronous Completion Token (ACT) design pattern. This design pattern efficiently dispatches processing within a client in response to the completion of asynchronous operations that the client invokes. For more information, see www.cs.wustl.edu/~schmidt/PDF/ACT.pdf.

When you use the ACT design pattern, you associate application-specific actions and state with responses that indicate the completion of asynchronous operations. For each asynchronous operation, you create an ACT that identifies the actions and state that are required to process the results of the operation. When the result is returned, you can use its ACT to distinguish it from the results of other asynchronous operations. The client uses the ACT to identify the state required to handle the result.

An ACT for a particular asynchronous operation is created before the operation is called. While the operation is executing, the client continues executing. When the service sends a response, the client uses the ACT associated with the response to perform the appropriate actions.

When you call a Flex remote object service, web service, or HTTP service, Flex returns an instance of the data service call. When you use the default `concurrency` value, `multiple`, you can use the call object returned by the data service's `send()` method to handle the specific results of each concurrent call to the same service. You can add information to this call object when it is returned, and then in a result event handler you can pass back the call object as `event.call`. This is an implementation of the ACT design pattern that uses the call object of each data service call as an ACT. How you use the ACT design pattern in your own code depends on your requirements. For example, you might attach simple identifiers to individual calls, more complex objects that perform their own set of functionality, or functions that a central handler calls.

The following example shows a simple implementation of the ACT design pattern. This example uses an HTTP service and attaches a simple variable to the call object.

```
...
<mx:HTTPService id="MyService" url="..." result="myHandler(event)" />
...
<mx:Script>
    <![CDATA[
        ...
        function storeCall()
        {
            // Create a variable called call to store the instance
            // of the service call that is returned.
            var call = MyService.send();

            // Add a variable to the call object that is returned.
            // You can name this variable whatever you want.
            call.marker = "option1";
            ...
        }

        // In a result event handler, execute conditional
        // logic based on the value of call.marker.
        function myHandler(event)
        {
            var call = event.call
            if (call.marker == "option1") {
                //do option 1
            }
            else
                ...
        }
    ]]>
</mx:Script>
...
```

Making a service call when another call is completed

Another common requirement when using data services is the dependency of one service call on the result of another. Your code must not make the second call until the result of the first call is available. You must make the second service call in the result handler of the first, as the following example shows:

```
...
<mx:WebService id="ws"...>
  <mx:operation name="getCurrentSales" result="myResultHandler(event.result)"
    />
  <mx:operation name="getForecastWithSalesInput" />
</mx:WebService>
<mx:Script>
  <![CDATA[
    // Call the getBarWithFoolInput operation with the result of the getFoo
    // operation.
    function myResultHandler(currentsales) {
      ws.getForecastWithSalesInput(currentsales);
      //Or some variation that uses data binding.
    }
  ]]>
</mx:Script>
...
```

Using callback URLs

Callback URLs are the URLs that are embedded in a SWF file to specify how to communicate with Flex on the server side for service objects. You need to manually specify callback URLs only if you precompile an application or want to use a custom location for the callback URLs because you are accessing data services from a computer other than the one serving the Flex application. By default, Flex bases callback URLs on the MXML request URL, and almost always handles them automatically. When necessary, you can manually specify a callback URL in the `flex-config.xml` file, by using the `flashVars` technique, or on the command line. The `flashVars` property of Macromedia Flash Player lets you import variables into the top level of an application when it is instantiated.

Callback URLs for remote object services

When you use remote object services, you specify callback URLs in the `<amf-gateway>` and `<amf-https-gateway>` tags in the `<remote-objects>` tag. The following example shows a `<amf-gateway>` tag set to a specific URL:

```
<remote-objects>
...
  <amf-gateway>{context.root}/amfgateway</amf-gateway>
...
</remote-objects>
```


You can also specify a callback URL by using flashVars variables. This technique is most useful during development for testing different proxy servers, but it is rarely needed in a production environment where setting up callback URLs in the flex-config.xml file is usually a better approach. Flex looks for flashVars variables named gatewayUrl or gatewayHttpsUrl. The flashVars technique works only when the <allow-url-override> is true for the <remote-objects> tag in the flex-config.xml file.

When you use the command-line compiler, you can specify a callback URL by using the gatewayurl or gatewayhttpsurl options. For more information about the command-line compiler, see [Chapter 36, “Administering Flex,” on page 795](#).

Callback URLs for web services and HTTP services

For web services and HTTP services, you can specify callback URLs in the flex-config.xml file in <url> and <https-url> tags in the <web-service-proxy> tag or <http-service-proxy>, respectively. The <url> and <https-url> tags are for HTTP proxy calls and HTTPS proxy calls. The following example shows a <url> tag set to a specific URL:

```
<web-service-proxy>
...
  <url>http://somesite.com/directory</url>
...
</web-service-proxy>
```

You can also specify a callback URL by using flashVars variables. This technique is most useful during development for testing different proxy servers, but it is rarely needed in a production environment where setting up callback URLs in the flex-config.xml file is usually a better approach. Flex looks for flashVars variables named proxyURL or proxyHttpsURL. The flashVars technique works only when the <allow-url-override> is set to true in the flex-config.xml file. For information about using flashVars variables, see [Chapter 38, “Deploying Applications,” on page 835](#).

When you are using the command-line compiler, you can specify a callback URL using the proxyurl and proxyhttpsurl options. For more information about the command-line compiler, see [Chapter 36, “Administering Flex,” on page 795](#).

Generating debugging information for data services

During development, you can generate debug information that can help you work with data services. The <debug> tag in the flex-config.xml file contains <remote-object-debug>, <web-service-proxy-debug>, and <http-service-proxy-debug> tags. When the value of one of these tags is true, debugging information is generated for that service type.

For remote object services, when you set the <remote-objects-debug> value to true, you can use the Net Connection Debugger to view information about AMF service calls and determine the structure of result objects. The Net Connection Debugger is in the extras/netConnectionDebugger directory of your Flex installation; to use it, open the NetConnectionDebugger.html file in a web browser.

For web services and HTTP services, debug information is added to logs on the client side in the flashlog.txt file and the server side in the console, and in the Flex log files in the WEB-INF/flex/logs directory of your Flex web application. On the server side, the request and response for these service types are printed out for easier debugging. This is very useful when you need to know the structure of a result object to correctly bind from it.

The following example shows a `<web-service-proxy-debug>` tag:

```
<debugging>
...
  <web-service-proxy-debug>false</web-service-proxy-debug>
...
</debugging>
```

Another useful tool for debugging data services is the `XMLObjectOutput` class, which lets you dump the content of objects so that you can understand their structure. For information about the `XMLObjectOutput` class, see the Flex TechNote [“Flex 1.0: Using XMLObjectOutput to dump the content of Objects”](#) on the Macromedia website.

Working with remote object services

You can use the `<mx:RemoteObject>` tag to call methods on Java objects that reside on the Java application server on which Flex is running. The `<mx:RemoteObject>` tag uses AMF, the encoding used in Macromedia Flash Remoting. AMF transport is faster and uses less network bandwidth than SOAP transport. You can call methods on Java objects in the Flex web application's classpath.

Java objects in the classpath

The `<mx:RemoteObject>` tag lets you access POJOs and JavaBeans that are in the web application's classpath. You can place stand-alone class files in the web application's WEB-INF/classes directory to add them to the classpath. You can place classes contained in Java Archive (JAR) files in the web application's WEB-INF/lib directory to add them to the classpath. You must specify the fully qualified class name in the `source` property. The class also must have a no-argument constructor.

Stateless objects

You use the `stateless-class` type, which is the default type, to create a new object for each method call instead of calling methods on the same object.

In the following example, each method call would be invoked on a new instance of the `credit.CreditCardAuth` class:

```
<mx:RemoteObject id="creditclass" type="stateless-class"
  source="credit.CreditCardAuth"/>
```

Stateful objects

When you use the `<mx:RemoteObject>` tag, you use the `stateful-class` type to call methods on a class on the server. When you access a Java object using this syntax, the class is loaded on the server and Flex maintains state between method calls. Use the `stateless-class` type if storing the object in the session causes memory problems.

For the following example, each method call is invoked on the same instance of the `credit.CreditCardAuth` class:

```
<mx:RemoteObject type="stateful-class" id="creditclass"
  source="credit.CreditCardAuth"/>
```

Flex uses J2EE server sessions to maintain the state of stateful Java objects. The server session feature depends on session cookies. The session cookie is sent automatically. If you want to ensure that non-cookie-handling clients have stateful access to Java objects, you can append the `jsessionid` to the URL. You typically do this in a JavaServer Pages (JSP) file by using the `response.encodeURL()` method.

The server session feature is available on Windows, Linux, and UNIX versions of the stand-alone Flash Player, but is not available on the stand-alone Flash Player for the Macintosh platform.

Enterprise JavaBeans and other objects in JNDI

You can access Enterprise JavaBeans and other objects stored in the Java Naming and Directory Interface (JNDI) by using a service facade class that looks up an object in JNDI and calls its methods.

Using a service facade class

When you use the `<mx:RemoteObject>` tag, you can use the `stateless-class` or `stateful-class` type to call the methods of Enterprise JavaBeans (EJBs) and other objects that use JNDI. For an EJB, you specify the source value as a service facade class that returns the EJB object from JNDI and contains a method that calls a method on the EJB.

In your Java class, you use the standard Java coding pattern, in which you create a new initial context and perform a JNDI lookup. For an EJB, you also use the standard coding pattern in which your class contains methods that call the EJB home object's `create()` method and the resulting EJB's business methods.

The following example uses a method called `getHelloData()` on a facade class called `HelloServiceClass`:

```
<mx:RemoteObject id="Hello" source="mypackage>HelloServiceClass">
  <mx:method name="getHelloData"/>
</mx:RemoteObject>
```

On the Java side, the `getHelloData()` method could easily encapsulate everything necessary to call a business method on an EJB. The Java method in the following example performs the following actions:

- Creates new initial context for calling the EJB
- Performs a JNDI lookup that gets an EJB home object

- Calls the EJB home object's `create()` method
- Calls the EJB's `sayHello()` method

```
...
getHelloData{
    try{
        InitialContext ctx = new InitialContext();

        Object obj = ctx.lookup("/Hello");

        HelloHome ejbHome = (HelloHome)

        PortableRemoteObject.narrow(obj, HelloHome.class);

        HelloObject ejbObject = ejbHome.create();

        String message = ejbObject.sayHello();
    }
    catch (Exception e);
}
...
```

Reserved method names

The Flex remote object library uses the following method names; do not use these as your own method names:

```
addHeader()
addProperty()
clearUsernamePassword()
deleteHeader()
hasOwnProperty()
isPropertyEnumerable()
isPrototypeOf()
registerClass()
setUsernamePassword()
toLocaleString()
toString()
unwatch()
valueOf()
watch()
```

You also should not begin method names with an underscore (`_`) character.

Converting data from ActionScript to Java

Data sent in method arguments from a Flex application to a Java object is automatically converted from an ActionScript data type to a Java data type. When Flex searches for a suitable method on the Java object, it uses further, more lenient conversions to find a match.

Simple data types on the client, such as Boolean and String, typically match a remote API exactly. However, Flex attempts some simple conversions when searching for a suitable method on a Java object.

An ActionScript Array can index entries in two ways. A strict Array is one in which all indexes are Numbers. An associative Array is one in which at least one index is based on a String. It is important to know which type of Array you are sending to the server because it will change the data type of arguments used to invoke a method on a Java object.

Simple data type conversions

The following table lists the supported ActionScript to Java conversions for simple data types:

| ActionScript | AMF deserialization | Supported Java data type conversions |
|---------------------|----------------------|---|
| null | null | null for Object, default values for primitives |
| Number (primitive) | java.lang.Double | java.lang.Number (class or primitive values), java.lang.String |
| Boolean (primitive) | java.lang.Boolean | java.lang.Boolean, boolean, java.lang.String |
| String (primitive) | java.lang.String | java.lang.String, java.lang.Boolean, boolean, java.lang.Character, char, java.lang.Number (class or primitive values) |
| Date | java.util.Date | java.util.Date |
| XML object | org.w3c.dom.Document | org.w3c.dom.Document |

If a Number is converted to a String to call a method on a Java object, it is represented as the result of the `java.lang.Double.toString()` API. For example, if the ActionScript Number 2 is sent to a method expecting a String, it is passed as “2.0”.

Primitive values cannot be set to null in Java. When passing values of type Boolean and Number from the client to a Java object, Flex interprets null values as the default for primitive types. For example, 0 for double, float, long, int, short, byte, `\u0000` for char, and false for Boolean.

Using the `new` operator in ActionScript to create an object representation of a Number, Boolean, or String is not supported with AMF and you cannot use it when sending data to a Java object. The code in the following example is not supported:

```
var id:Number = new Number(14); //Not supported.
MyService.getProduct(id);
```

Strict Arrays

The following table lists the ActionScript to Java conversions for strict ActionScript Arrays:

| ActionScript | AMF Deserialization | Supported Java data type conversions |
|----------------------|---------------------|--------------------------------------|
| Array (ordinal keys) | java.util.ArrayList | Collection, Object[] |

You can pass strict Arrays as arguments to methods expecting an implementation of the `java.util.Collection` or `java.lang.reflect.Array` APIs.

A Java Collection can contain any number of Object types, where as a Java Array requires that entries are the same type (for example, `java.lang.Object[]`, and `int[]`).

Flex also converts ActionScript strict Arrays to appropriate implementations for common Collection API interfaces. For example, if an ActionScript strict Array is sent to the Java object method `public void addProducts(java.util.Set products)`, Flex converts it to a `java.util.HashSet` instance before passing it as an argument, because `HashSet` is a suitable implementation of the `java.util.Set` interface. Similarly, Flex passes an instance of `java.util.TreeSet` to arguments typed with the `java.util.SortedSet` interface. Flex passes an instance of `java.util.ArrayList` to arguments typed with the `java.util.List` interface and any other interface extending `java.util.Collection`.

Associative Arrays and untyped Objects

The following table lists the supported ActionScript to Java conversions for associative ActionScript Arrays and untyped Objects:

| ActionScript | AMF deserialization | Supported Java data type conversions |
|--------------------------|--------------------------|--------------------------------------|
| Array (associative keys) | flashgateway.io.ASObject | Map |
| Object (untyped) | flashgateway.io.ASObject | Map |

The server treats associative Arrays (an Array with at least one String-based index) and untyped Objects from the client as general structures of key-value pairs. Although the keys from these client types are always sent as Strings, for historical reasons, `flashgateway.io.ASObject`, an implementation of the `java.util.Map` interface, is used to represent these structures.

If a concrete implementation of the `Map` interface is required for a method argument, the contents of the `ASObject` are copied into a `Map` object.

The server also converts these structures to appropriate implementations for common Map API interfaces. For example, if an untyped ActionScript Object is sent to the Java object method `public void addDescriptions(java.util.SortedMap descriptions)`, it is converted to a `java.util.TreeMap` instance before Flex passes it as an argument, because `TreeMap` is a suitable, standard implementation of `java.util.SortedMap`. Similarly, Flex passes an instance of `java.util.HashMap` to arguments typed with the `java.util.Map` interface.

Typed Objects

The following table lists the supported ActionScript to Java conversions for typed Objects:

| ActionScript | AMF deserialization | Supported Java data type conversions |
|----------------|--------------------------|--------------------------------------|
| Object (typed) | flashgateway.io.ASObject | new CustomType() |

Objects that are not handled implicitly

When an ActionScript type is not handled implicitly, you can map it to a typed Java class of the same name on the server. Create a static variable in the ActionScript class that uses the `Object.registerClass()` method to specify the fully qualified name of the corresponding Java class on the server. The first argument of the `registerClass()` method is the fully qualified name of the Java class; the second argument is the fully qualified name of the ActionScript class. If Flex finds a class with the same name, Flex uses public variables and JavaBean-like setters to set values on the object. Before setting these values, Flex also tries to convert these objects to their implicit or class matching types.

Note: The package of the Java class on the server must be included in the unnamed whitelist for remote objects in the `flex-config.xml` file.

Your ActionScript class cannot use properties that are declared as getter/setter properties using the `get` and `set` keywords; the properties must be real variables.

The following example shows a simple ActionScript class that uses the `Object.registerClass()` method to create a mapping to a Java class in the `regClass` variable:

```
class com.Product
{
    public var id:Number;
    public var name:String;
    public var price:Number;
    public var description:String;
    public static var regClass = Object.registerClass("com.Product",
        com.Product);
    function Product()
    {
    }
    function toString()
    {
        return "id = " + id + " name = " + name + " price = $" + price;
    }
}
```

The following example shows the corresponding Java class on the server:

```
package com;
public class Product {
    public Integer id;
    public String name;
    public double price;
    public String description;

    public Product()
    {
    }
}
```

If you do not specify a matching class on the server, Flex uses an `ASObject` as the corresponding server-side object; `ASObject` extends `HashMap`. Before setting these values, Flex attempts to convert objects to their implicit or class matching type.

Flex sends private variables from complex ActionScript objects to the server side. To hide a variable, add the following code to your object's constructor, substituting your property's name for *propertyname*:

```
_global.ASSetPropFlags(this,"propertyname",1);
```

Converting data from Java to ActionScript

An object returned from a Java method is converted from Java to ActionScript. Flex also handles objects found within objects. Flex handles the following Java data types implicitly:

| Java | ActionScript |
|--|------------------|
| null | null |
| java.lang.Number (class or primitive) | Number |
| java.lang.Boolean, boolean | Boolean |
| java.lang.String, java.lang.Character, char[] | String |
| java.util.Collection, java.lang.reflect.Array | Array |
| java.util.Map, java.util.Dictionary, java.lang.Throwable | Object (untyped) |
| org.w3c.dom.Document | XML object |
| Other classes | Object (typed) |

For Java objects that Flex does not handle implicitly, values found in JavaBean-like getter methods and public variables are passed to the client as properties on an object.

Flex automatically looks for an ActionScript class based on the Java class name. In the ActionScript class, you use the `Object.registerClass()` method to create a mapping to a Java class, as described in [“Converting data from ActionScript to Java” on page 676](#). The ActionScript class to which data is to be converted must be used or referenced in the MXML file. A good way to do this is by casting the result object, as the following example shows:

```
var result:MyClass = MyClass(event.result);
```

The class itself should use strongly-typed references so that its dependencies are also linked.

Your ActionScript class cannot use properties that are declared as getter/setter properties using the `get` and `set` keywords; the properties must be real variables.

Note: When registering a class, the NetConnection Debugger does not understand the data type, and reports arguments of this type as undefined. You can ignore this. Data is still sent and received correctly.

Working with session data

A Java object that you call using the `<mx:RemoteObject>` tag has access to request, response, and servlet data. From within a Java object, you can call the following methods:

| Method | Description |
|--|---|
| <code>flashgateway.Gateway.getHttpRequest()</code> | Returns the <code>HttpServletRequest</code> object for the current request. Macromedia recommends that you access session data and other request data through the <code>getHttpRequest()</code> method. |
| <code>flashgateway.Gateway.getHttpResponse()</code> | Returns the <code>HttpServletResponse</code> object for the current request. |
| <code>flashgateway.Gateway.getServletConfig()</code> | Returns the <code>ServletConfig</code> object for the calling servlet. |

To compile calls with these methods in their classes, you must have the `WEB-INF/lib/flashgateway.jar` file in your classpath.

The following example shows code in a Java class for accessing a session attribute:

```
String fooAttrib = (String)flashgateway.Gateway.getHttpRequest().getSession().
    getAttribute("attr1");
```

Working with web services

Flex applications can interact with web services that define their interfaces in a Web Services Description Language (WSDL) document, which is available as a URL. WSDL is a standard format for describing the messages that a web service understands, the format of its responses to those messages, the protocols that the web service supports, and where to send messages.

Note: Flex does not support the following XML schema types: choice, union, default, list, or group. duration. Flex also does not support the following data types: date, gMonth, gYear, gYearMonth, gDay, gMonthDay, Name, QName, NCName, anyURI, or language.

Flex applications support web service requests and results that are formatted as Simple Object Access Protocol (SOAP) messages and are transported over Hypertext Transfer Protocol (HTTP). SOAP provides the definition of the XML-based format that you can use for exchanging structured and typed information between a web service client, such as a Flex application, and a web service.

Flash Player operates within a security sandbox that limits what Flex applications and other Macromedia Flash applications can access over HTTP. Flash applications are only allowed HTTP access to resources on the same domain and by the same protocol from which they were served. This presents a problem for web services, because they are typically accessed from remote locations. The Flex proxy intercepts requests to remote web services, redirects the requests, and then returns the responses to the client.

Reading WSDL documents

You can view a WSDL document in a web browser, a simple text editor, an XML editor, or a development environment such as Macromedia Dreamweaver MX, which contains a built-in utility for displaying WSDL documents in an easy-to-read format.

A WSDL document contains the tags described in the following table:

| Tag | Description |
|-------------|--|
| <binding> | Specifies the protocol clients, such as Flex applications, use to communicate with a web service. Bindings exist for SOAP, HTTP GET, HTTP POST, and MIME. Flex supports the SOAP binding only. |
| <fault> | Specifies an error value returned as a result of a problem processing a message. |
| <input> | Specifies a message that a client, such as a Flex application, sends to a web service. |
| <message> | Defines the data that a web service operation transfers. |
| <operation> | Defines a combination of <input>, <output>, and <fault> tags. |
| <output> | Specifies a message that the web service sends to a web service client, such as a Flex application. |
| <port> | Specifies a web service endpoint, which specifies an association between a binding and a network address. |
| <portType> | Defines one or more operations that a web service provides. |
| <service> | Defines a collection of <port> tags. Each service maps to one <portType> tag and specifies different ways to access the operations in that <portType> tag. |
| <types> | Defines data types that a web service's messages use. |

RPC-oriented operations and document-oriented operations

A WSDL file can specify either remote procedure call (RPC) oriented or document-oriented (document/literal) operations. Flex supports both operation styles.

When calling an RPC-oriented operation, a Flex application sends a SOAP message that specifies an operation and its arguments. When calling a document-oriented operation, a Flex application sends a SOAP message that contains an XML document. It is important that you know how to identify these operation styles in a WSDL document, because they can affect the tags that you use in a Flex application.

In a WSDL document, each <port> tag has a binding property that specifies the name of a particular <soap:binding> tags, as the following example shows:

```
<binding name="InstantMessageAlertSoap" type="s0:InstantMessageAlertSoap">  
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"  
    style="document" />
```

The style property of the associated <soap:binding> tag determines the operation style. In this example, the style is document.

Any operation in a service can specify the same style or override the style specified for the port associated with the service, as the following example shows:

```
<operation name="SendMSN">
  <soap:operation soapAction="http://www.bindingpoint.com/ws/imalert/SendMSN"
    style="document"/>
```

Stateful web services

Flex uses Java server sessions to maintain the state of web service endpoints that use cookies to store session information. This feature acts as an intermediary between Flex applications and web services. It adds an endpoint's identity to whatever the endpoint passes to a Flex application. If the endpoint sends session information, the Flex application receives it. This feature requires no configuration. Flex detects whether a `jsessionid` is required and automatically updates the AMF gateway URL accordingly.

Working with SOAP headers

A SOAP header is an optional tag in a SOAP envelope that usually contains application-specific information, such as authentication information or payment information. This section describes how to add SOAP headers to web service requests, clear SOAP headers, and get SOAP headers contained in web service results.

Adding SOAP headers to web service requests

Some web services require that you pass along a SOAP header when you call an operation.

To add a SOAP header to web service calls, you can use a `WebService` object's `addHeader()` method or `addSimpleHeader()` method in an event handler function.

When you use the `addHeader()` method, you first must create `SOAPHeader` and `QName` objects separately. The `addHeader()` method has the following signature:

```
public function addHeader(header : mx.services.SOAPHeader) : Void
```

To create a `SOAPHeader` object, you use the following method signature:

```
public function SOAPHeader(qname : mx.services.QName, content)
```

To create the `QName` object in the first argument of the `SOAPHeader()` method, you use the following method signature:

```
public function QName(localPart, namespaceURI)
```

The `content` argument of `SOAPHeader()` method is a set of name-value pairs based on the following format:

```
{name: value, name2: value2}
```

The `addSimpleHeader()` method is a shortcut for a single name-value SOAP header. When you use the `addSimpleHeader()` method, you create `SOAPHeader` and `QName` objects in arguments of the method. The `addSimpleHeader()` method has the following signature:

```
public function addSimpleHeader(qnameLocal : String, qnameNamespace : String,
  headerName : String, headerValue) : Void
```

The `addSimpleHeader()` method takes the following arguments:

- `qnameLocal` is the local name for the header `QName`.
- `qnameNamespace` is the namespace for header `QName`.
- `headerName` is the name of the header.
- `headerValue` is the value of the header. This can be a `String` if it is a simple value, an `Object` that will undergo basic XML encoding, or XML if you want to specify the header XML yourself.

The code in the following example shows how to use the `addHeader()` method and the `addSimpleHeader()` method to add a SOAP header. The methods are called in an event handler function called `headers`, and the event handler is assigned in the `load` property of an

`<mx:WebService>` tag:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" height="600" >

    <!-- The value of the wsdl property is for demonstration only and is not a
    valid WSDL document. -->

    <mx:WebService id="ws" wsdl="@ContextRoot()/test.wsdl" load="headers();" />
    ...
    <mx:Script>
        <![CDATA[
            var header1;

            function headers() {

                /* Create QName and SOAPHeader objects. */
                var q1=new mx.services.QName("Header1", "http://soapinterop.org/xsd");
                header1=new mx.services.SOAPHeader(q1, {string:"bologna",int:"123"});

                /* Add the header1 SOAP Header to web service requests. */
                ws.addHeader(header1);

                /* Within the addSimpleHeader method, which adds a SOAP header to web
                service requests, create SOAPHeader and QName objects. */
                ws.addSimpleHeader("Header2", "http://soapinterop.org/xsd", "foo",
                "bar");
            }
        ]]>
    </mx:Script>
    ...
</mx:Application>
```

Clearing SOAP headers

You use a `WebService` object's `clearHeaders()` method to remove SOAP headers that you added to a `WebService` object, as the following example shows:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" height="600" >
```

```

<!-- The value of the wsdl property is for demonstration only and is not a
real WSDL document. -->

<mx:WebService id="ws" wsdl="@ContextRoot()/test.wsdl" load="headers();" />
...
<mx:Script>
  <![CDATA[
    var header1;

    function headers() {

      /* Create QName and SOAPHeader objects. */
      var q1=new mx.services.QName("Header1", "http://soapinterop.org/xsd");
      header1=new mx.services.SOAPHeader(q1, {string:"bologna",int:"123"});
      /* Add the header1 SOAP Header to web service requests. */
      ws.addHeader(header1);

      /* Within the addSimpleHeader method, which adds a SOAP header to web
      service requests, create SOAPHeader and QName objects. */
      ws.addSimpleHeader("Header2",
        "http://soapinterop.org/xsd", "foo", "bar");
    }

    /* Clear SOAP headers. */
    function clear(){
      ws.clearHeaders();
    }

  ]]>
</mx:Script>
...
<mx:HBox>
  <mx:Button label="Clear headers and run again"
  click="clearAndRetry();"/>
</mx:HBox>

...
</mx:Application>

```

Handling SOAP headers returned in SOAP responses

To handle custom SOAP headers that are returned on the SOAP response, you attach an `onHeaders()` function to the `PendingCall` object. You create a function to handle the headers, such as the `handleHeaders()` function in the following example:

```

<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" height="600" >
  <mx:WebService id="ws"
    wsdl="http://www.whitemesa.net/wsdl/r3/interopstestheaders.wsdl"
    load="postLoad()">
    <mx:operation name="echoString"
      result="call=event.call;echoStringResult(event.result);"
      fault="echoStringFault(event.fault);" />
  
```

```

</mx:WebService>

<mx:Script>
  <![CDATA[
    function echoStringResult(result) {
      trace(result);
    }
    function echoStringFault(fault) {
      trace(fault.faultstring);
    }

    function handleHeaders(responseHeaders, wholeResponse) {
      trace("**" + responseHeaders);
      var i = 0;
      viewHeaders.text = responseHeaders;
    }
    function postLoad() {
      var qName=new mx.services.QName("Header1",
        "http://soapinterop.org/xsd");
      var header1=new mx.services.SOAPHeader(qName,
        {string:"bologna",int:"123"});
      ws.addHeader(header1);
      ws.addSimpleHeader("Header2",
        "http://soapinterop.org/xsd", "foo", "bar");
      var call = ws.echoString("foo");
      call.onHeaders = mx.utils.Delegate.create(this, handleHeaders);
    }
  ]]>
</mx:Script>
  <mx:TextArea id="viewHeaders" height="300" width="300"/>
</mx:Application>

```

The `handleHeaders()` function takes two arguments: `responseHeaders` and `wholeResponse`. The `responseHeaders` argument is an XML object that contains just the SOAP headers. The `wholeResponse` argument is an XML object that contains the whole response. The `handleHeaders()` function is called when the response comes back from the web service. You then must manually parse the XML in the `responseHeaders` object and do any custom handling of the SOAP headers.

When the application calls a web service operation, it returns a `PendingCall` object. In this example, this object is assigned to a variable called `call`, and the `onHeaders()` function is assigned to that variable. The `call` variable references the `handleHeaders()` function. To avoid scoping issues, the `handleheaders()` function is wrapped in a call to the `mx.utils.Delegate.create()` method.

Redirecting a web service to a different URL

Some web services require that you change to a different endpoint URL after you process the WSDL and make an initial call the web service. For example, suppose you want to use a web service that requires you to pass security credentials. After you call the web service to send login credentials, it accepts the credentials and returns the actual endpoint URL required to use the service's business operations. Before calling the business operations, you must change the `endpointURI` property of your `WebService` instance.

The following example shows a result event handler that stores the endpoint URL that a web service returns in a variable, and then passes that variable into a function to change the endpoint URL for subsequent requests:

```
function onLoginResult(event){

    // Extract the new service endpoint from the login result.
    var newServiceURL = event.result.serverUrl;

    // Redirect all service operations to the URL received in the login result.
    serviceName.setEndPointURI(newServiceURL);

}
```

A web service that requires you to pass security credentials, might also return an identifier that you must attach in a SOAP header for subsequent requests; for more information, see [“Working with SOAP headers” on page 683](#).

Securing data services

This section describes how to configure data service access and authentication.

Configuring whitelists to control access to services

To prevent the AMF gateway and the Flex proxy and from being used to stage denial of service (DOS) attacks or allow unauthorized access to services, Flex uses *whitelists*. By default, access to services is denied, and you must explicitly allow access.

Configuring remote object whitelists

You configure a Flex application's access to the Java classes used with unnamed and named remote object services in the remote objects whitelist in the `flex-config.xml` file. For more information about unnamed and named services, see [“Declaring a data service” on page 658](#).

You specify access to Java classes in `<source>` tags. By default, access to all classes is denied, and you must explicitly allow access. You can specify the fully qualified class name or a wildcard pattern; the `<source>` tag is case-sensitive. For example, you can use a wildcard asterisk (*) by itself, which indicates that all Flex classes and system classes are accessible; this type of configuration is useful for development, but you should always limit access to specific classes when in production.

The following example shows the `<whitelist>` tag for remote object services in a `flex-config.xml` file. The value of the `<source>` tag in the unnamed whitelist allows access to all classes in a package named `myPackage`. The value of the `<source>` tag in the named whitelist allows access to just the `samples.explorer.SalaryManager` class and its alias `MyService`.

```
<remote-objects>
...
  <whitelist>
    ...
    <unnamed>
      <source>myPackage.*</source>
    </unnamed>
    ...

    <named>
      <object name="MyService">
        <source>samples.explorer.SalaryManager</source>
        <type>stateful-class</type>
        <use-custom-authentication>true</use-custom-authentication>
      </object>
    </named>
  </whitelist>
...
</remote-objects>
```

Configuring web service and HTTP service whitelists

A whitelist is a list of URLs to which the administrator explicitly gives access to the Flex proxy. For both named and unnamed services, only the URLs explicitly allowed by the Flex administrator are accessible.

You add unnamed and named services to the `<whitelist>` tag for web services or HTTP services in the `flex-config.xml` file. The following example shows the `<whitelist>` tag for web services in a `flex-config.xml` file:

```
<web-service-proxy>
...
  <whitelist>
    ....
    <unnamed>
      <url>http://xmethods.net/services/service1</url>
      <url>http://myServer.com/services/*</url>
    </unnamed>
    ...

    <named>
      <service name="ContactManagerWS">
        <wsdl>{context.root}/services/ContactManagerWS?wsdl</wsdl>
        <endpoints>
          <endpoint>
            {context.root}/services/ContactManagerWS
          </endpoint>
        </endpoints>
        <use-custom-authentication>true</use-custom-authentication>
      </service>
    </named>
  </whitelist>
</web-service-proxy>
```



```

        </service>
    </whitelist>
    ...
</web-service-proxy>

```

The URLs in the unnamed whitelist apply to all unnamed services. You can use wildcards in unnamed whitelists. For example, adding the following URL allows access to all URLs that start with `http://`:

```
<url>http://*</url>
```

Note: This type of configuration is useful for development, but you should always limit access to specific URLs when in production. By default, access to all URLs is denied, and you must explicitly specify access.

To add approved URLs for a named services, you define the services using a `<service>` child tag of the `<web-service-proxy>` tag or the `<http-service-proxy>` tag, respectively. Flex adds the URLs to the whitelist for you.

Configuring authentication

The AMF gateway provides a mechanism for using remote object services that require the client to be authenticated. The Flex proxy provides mechanisms for using web services and HTTP services that require the client to be authenticated. This section describes these mechanisms.

Note: The `resources/security/examples` directory of your Flex installation contains examples of authentication.

Configuring authentication for remote object services

You can only configure authentication for named services. There are two types of authentication: Basic and custom.

Basic authentication

One way to authenticate users is to use Basic authentication. Basic authentication uses standard J2EE security and depends on security constraints for URLs that you set up in the `web.xml` file.

When you use named services with the `<mx:RemoteObject>` tag, Flex appends the service name to the URL that it uses to contact the remote service. This gives you a unique URL to use in a resource constraint. The objects that you access use the following URL pattern:

```
/amfgateway/MyService
```

The *MyService* part of the URL contains the name specified in the `name` property of the `<object>` tag. By default, the base portion of the URL corresponds to the value of the `<amf-gateway>` tag in the `flex-config.xml` file, which matches the servlet mapping for the AMF gateway in the `web.xml` file.

The settings for each named remote object service in the `flex-config.xml` file are in an `<object>` tag. An `<object>` tag set up for Basic authentication contains a `<use-basic-authentication>` tag that is set to `true`. The following example shows a named service set up to use Basic authentication:

```
<remote-objects>
...
  <whitelist>
    ...
    <named>
      <object name="MyService">
        <source>samples.explorer.SalaryManager</source>
        <type>stateful-class</type>
        <use-basic-authentication>true</use-basic-authentication>
      </object>
    </named>
    ...
  </whitelist>
...
</remote-objects>
```

In the `web.xml` file for your web application, you set up `security-constraint` and `login-config` sections to protect the `/amfgateway/MyService` URL pattern, as the following example shows.

```
<!-- Within the web-app tag: -->
<web-app>
...
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Protected Page</web-resource-name>

      <!-- Use this URL pattern for RemoteObject. -->
      <url-pattern>
        /amfgateway/MyService
      </url-pattern>

      <http-method>GET</http-method>
      <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
      <role-name>manager</role-name>
    </auth-constraint>
  </security-constraint>

  <login-config>
    <auth-method>BASIC</auth-method>
  </login-config>
...
</web-app>
```

You also must link the specified role references to roles defined for your application server. How you define users and roles is specific to your application server. For example, by default, you define Macromedia JRun 4 users and roles in the `servers/server_name/SERVER-INF/jrun-users.xml` file. For more information, see your application server documentation and the `addingusers.txt` files in the `resources/security/examples` directory of your Flex installation.

When you use a named service in an `<mx:RemoteObject>` tag, you specify the name of the service in a `name` property, as the following example shows:

```
<mx:RemoteObject id="remoteService" name="MyService"/>
```

When a Flex application accesses `http://yourdomain.com/yourapplication/amfgateway/MyService`, it must be authenticated for the session before being allowed to access the AMF gateway and the target remote object.

Custom authentication

As an alternative to Basic authentication, you can use custom authentication and create a custom login form in MXML to match the appearance of your application.

Unlike Basic authentication, custom authentication requires a login command, which is a Java class that implements the `flashgateway.security.LoginCommand` interface. Flex includes login command implementations for Macromedia JRun, BEA WebLogic, IBM WebSphere, and Apache Tomcat.

Note: The Tomcat login command requires special configuration. For more information, see the `readme.txt` file in the `resources/security/TomcatLogin` directory of your Flex installation.

You can use a login command without roles for custom authentication, but if you also want to use custom authorization, you must link the specified role references to roles that are defined for your application server. How you define users and roles is specific to your application server. For example, by default, you define Macromedia JRun 4 users and roles in the `servers/server_name/SERVER-INF/jrun-users.xml` file. For more information, see your application server documentation and the `addingusers.txt` files in the `resources/security/examples` directory of your Flex installation.

For an application server for which Flex does not include a login command implementation, you can create a custom login command. For more information, see the `readme.txt` file in the `resources/security/CustomLoginCommand` directory of your Flex installation.

Note: Previous versions of Flex supported the SOAP protocol with the `<mx:RemoteObject>` tag. With SOAP, you were required to set up a resource-constraint in the `web.xml` file for the resource. Do not do this when using custom authentication with AMF. Instead, use a login command to log the user into the J2EE server. Also, note that when using AMF you set up roles in the `flex-config.xml` file instead of the `web.xml` file.

The settings for each named remote object service in the `flex-config.xml` file are in an `<object>` tag. An `<object>` tag set up for custom authentication contains a `<use-custom-authentication>` tag set to `true`, and a `<roles>` tag. You specify the roles of users who are allowed to access the service in the `<roles>` tag. In the following example, only users assigned to the role `sampleuser` or `admin` can access the named service:

```
<remote-objects>
...
  <whitelist>
    ...
    <named>
      <object name="CustomAuthSampleSalaryRO">
        <source>samples.explorer.SalaryManager</source>
        <type>stateful-class</type>
        <use-custom-authentication>true</use-custom-authentication>
        <roles>
          <role>sampleusers</role>
          <role>admin</role>
        </roles>
      </object>
    </named>
    ...
  </whitelist>
...
</remote-objects>
```

Note: If you specify type values in the `flex-config.xml` file, you do not need to specify them in the `<mx:RemoteObject>` tag. If you specify type values in both places, the values must match or an error is reported.

If Flex doesn't find suitable credentials, it throws an exception with the status code `Client.Authentication`. You can check this in the fault handler assigned in your `<mx:RemoteObject>` tag and prompt the user for credentials using the `setUsernamePassword()` method, as the following example shows:

```
MyService.setUsernamePassword("username", "password");
```

You can clear credentials using the `clearUsernamePassword()` method call, as the following example shows:

```
MyService.clearUsernamePassword();
```

The following example shows a fault handler that checks for the `Client.Authentication` fault code.

```
function faultHandler(fault) {
  if (fault.faultcode == "Client.Authentication") {
    // User is not authenticated -> display a logon window
    var popup = mx.managers.PopUpManager.createPopUp(this, LogonWindow, true,
      {deferred: true});
    popup.move(120,100);
    popup.addEventListener("logonWindowEvent", this);
  } else {
    alert(fault.faultcode + ": " + fault.faultstring, "Remote Object Error");
  }
}
```

Note: When the user name and password values are cleared, the user is still authorized to view the page. This is because of the way that servlet security works. Users are tracked by their `jsessionId`, and they do not need to re-authenticate themselves as long as that `jsessionId` is valid.

Configuring authentication for web service and HTTP services

You can only configure authentication for named services. There are three options for handling a service endpoint URL that is secured using Basic authentication: pass-through, custom, and run-as authentication. Each of these options only works when the endpoint URL is secured using Basic authentication.

You can also secure the Flex proxy itself using standard J2EE security.

Pass-through authentication

By default, Flex uses pass-through authentication. When an endpoint resides on the same server as Flex and the endpoint URL is secured, the Flex proxy passes back the authentication challenge to the client, which triggers the web browser to display its user name and password dialog box. This only works when the endpoint URL and Flex are on the same server.

Custom authentication

When you set the `<use-custom-authentication>` tag to `true` the Flex proxy intercepts authentication errors and returns them to the client as faults. For web services, the fault value is the fault code `Client.Authentication`. For HTTP services, the fault code is `401`. The client code can use this fault code to display a more attractive or informative dialog box that is consistent with your MXML application's user interface. After you obtain user name and password credentials, you can attach them to all future requests for that service using the `setUsernamePassword()` method, as the following example shows:

```
MyService.setUsernamePassword("username", "password");
```

Note: The user name and password data is not sent using the standard Authorization header; instead, it is sent to the Flex proxy, which creates the header. For HTTP services, you must use the HTTP POST method with the `setUsernamePassword()` method.

You can clear credentials using the `clearUsernamePassword()` method call, as the following example shows:

```
MyService.clearUsernamePassword();
```

When the user name and password values are cleared, the user is still authorized to view the page, even if you call the `service.clearUsernamePassword()` method. This is because of the way that servlet security works. Users are tracked by their `jsessionId`, and they do not need to re-authenticate themselves as long as that `jsessionId` is valid. Clearing the user name and password is useful if you have session access turned off from the service that you are contacting, or you log off the user on the server in a manner specific to the application server.

The following example shows a fault handler for an HTTP service that checks for the `401` fault code. A fault handler for a web service should check for the `Client.Authentication` fault code.

```
function faultHandler(fault) {  
    if (fault.faultcode == "401") {  
        // User is not authenticated -> display a logon window
```

```

        var popup = mx.managers.PopUpManager.createPopUp(this, LogonWindow, true,
            {deferred: true});
        popup.move(120,100);
        popup.addEventListener("logonWindowEvent", this);
    } else {
        alert(fault.faultcode + ": " + fault.faultstring, "HTTP Service Error");
    }
}

```

Note: You cannot use an unsecured WSDL URL with a secured web service when using Basic authentication. Instead, you must secure the WSDL URL or use custom authentication.

Run-as authentication

To automatically authenticate users without displaying a login dialog box, you can specify a user name and password for a named service in a `<run-as>` tag, as the following example shows:

```

<web-service-proxy>
...
  <whitelist>
    ...
    <named>
      <service name="MyService">
        <wsdl>http://somewhere.com/webservice.wsdl</wsdl>
        <endpoint>http://somewhere.com/myservice</endpoint>
        <run-as user="user1" password="opensaysme"/>
      </service>
    </named>
    ...
  </whitelist>
...
</web-service-proxy>

```

Flex applies the specified user name and password to service requests. If these credentials result in an HTTP 401 error (authentication required), the application uses pass-through or custom authentication, depending on how the service is configured. The run-as information resides on the server and is never passed to the client. The run-as information overrides whatever you send from the client using the `setUsernamePassword()` and `clearUsernamePassword()` methods.

Securing the Flex proxy URL

In addition to setting up authentication for secure services, you can secure the Flex proxy URL itself by using standard J2EE security. This is the only case in which the application server on which Flex is hosted controls security.

You set up `<security-constraint>` and `<login-config>` sections in the `web.xml` file for your web application to protect the `/flashproxy/MyService` URL pattern, as the following example shows:

```

<web-app>
...
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Protected Page</web-resource-name>
      <url-pattern>/flashproxy/MyService</url-pattern>
    </web-resource-collection>
  </security-constraint>

```

```

        <http-method>GET</http-method>
        <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint>
        <role-name>manager</role-name>
    </auth-constraint>
</security-constraint>

<login-config>
    <auth-method>BASIC</auth-method>
</login-config>
...
</web-app>

```

When Flex and the service endpoint are on different servers and the Flex proxy is secured, the browser displays a username and password dialog the first time that the application attempts to access the proxy. This happens before any contact is made to the endpoint. The user must log in to continue. After the user logs in, contact is made to the endpoint. If the endpoint is secured, it returns a 401 code to the proxy, which the proxy must decide what to do with.

In this scenario, you cannot use pass-through authentication. If run-as credentials are set, the proxy tries to use them. If the `<use-custom-authentication>` tag is set to `true`, the 401 code is returned as a `Client.Authentication fault`.

When Flex and the endpoint are on same server and the proxy is secured, a username and password dialog the first time that the application attempts to access the proxy. This happens before any contact is made to the endpoint. The user must log in to continue. After the user logs in, contact is made to the endpoint. The proxy passes the credentials that were used to log in along with service calls (pass-through authentication). If the endpoint is also secured and the credentials are valid for it, the service should work. If the credentials are not valid for the endpoint, you can use run-as or custom authentication. If run-as credentials are set, the proxy tries to use them. If the `<use-custom-authentication>` tag is set to `true`, the 401 code is returned as a `Client.Authentication fault`.

In this scenario, security would most likely be on the whole application, so pass-through authentication would typically work and no additional work would be required. A slight variation on this would be when the whole application is secured, but the services are secured with different login requirements so users can't access them directly. In that case, you could provide the required login information as run-as credentials.

Using HTTPS

Clients can access the Flex proxy using HTTP or HTTPS. The AMF gateway and Flex proxy can connect to HTTP and HTTPS URLs. If a Flex application is served over HTTPS, it can access HTTP or HTTPS services. No special configuration is required for accessing HTTPS services.

To access a data service over HTTPS from a Flex application that is served over HTTP, you must set the `protocol` property of the `<mx:RemoteObject>` tag, `<mx:HTTPService>` tag, or `<mx:WebService>` tag to `https`. For remote object services, you must set the `<amf-https-gateway>` tag in the `<remote-objects>` tag of the `flex-config.xml` file to the absolute HTTPS URL of the AMF gateway. For HTTP services and web services, you must set the `<https-url>` tag in the `<http-service-proxy>` tag or `<web-service-proxy>` tag to the absolute HTTPS URL of the Flex proxy.

By default, Flash Player does not allow an application loaded through HTTP to make requests through HTTPS, even for the same domain. To allow these requests, you must set the `secure` property of the `<cross-domain-policy>` tag to `false` in the `crossdomain.xml` file, as the following example shows:

```
<cross-domain-policy>
  <allow-access-from domain="*" secure="false" />
</cross-domain-policy>
```

The `crossdomain.xml` file lets you override the default behavior of the Flash security sandbox.

Note: The `crossdomain.xml` file must be in the web root of the server that the Flex application is contacting. If an HTTP service or web service does not go through the proxy (either the `useProxy` property in the `<mx:HTTPService>` or `<mx:WebService>` tag is set to `false` or the `<proxy-use-policy>` tag in the `<http-service-proxy>` tag or `<web-service-proxy>` tag in the `flex-config.xml` file is set to `never`), the `crossdomain.xml` file must be on the endpoint server. For more information about the `crossdomain.xml` file, see [Chapter 37, “Applying Flex Security,” on page 823](#).

Data service tag properties

This section describes the properties that are common to the `<mx:RemoteObject>`, `<mx:WebService>`, and `<mx:HTTPService>` tags, and the properties that are specific to each tag.

Common data service properties

The `<mx:RemoteObject>`, `<mx:WebService>`, and `<mx:HTTPService>` tags have the following properties in common:

| Property | Description |
|-----------------------------|--|
| <code>concurrency</code> | <p>The default value for the <code>concurrency</code> property. This property indicates how to handle multiple calls to the same method. The following values are permitted:</p> <ul style="list-style-type: none"><code>single</code> Only a single request is allowed on the operation; multiple requests generate a fault.<code>last</code> Making a request cancels any existing request.<code>multiple</code> Existing requests are not cancelled, and the developer is responsible for ensuring the consistency of returned data by carefully managing the event stream. This is the default value. |
| <code>fault</code> | <p>The default value for the handler for the <code>fault</code> event. This property specifies ActionScript code that runs when an error occurs.</p> <p>When no event handler is specified for faults at the web service operation level or remote object service method level, the faults are passed to the <code>fault</code> event at the service level; for more information, see “Handling asynchronous calls to data services” on page 670.</p> <p>The following example tag specifies a handler for the <code>fault</code> event of a web service:</p> <pre><!-- The specified WSDL URL is not functional. --> <mx:WebService id="myWebService" wsdl="/ws/WeatherService?wsdl" fault="showErrorDialog(event.fault.faultstring)"/></pre> |
| <code>id</code> | <p>The instance name that you use to refer to the web services, HTTP service, or remote object service in ActionScript code.</p> |
| <code>protocol</code> | <p>Specifies the protocol to use for service requests. The value is either <code>http</code> or <code>https</code>. The default value is the protocol over which the user loads the application.</p> <p>To access a service over HTTPS from a Flex application that is served over HTTP, you must set the <code>protocol</code> property to <code>https</code>.</p> |
| <code>result</code> | <p>The default value for the handler for the <code>result</code> event. This property specifies ActionScript code that runs when a result object is available.</p> |
| <code>showBusyCursor</code> | <p>When set to <code>true</code>, a busy cursor is displayed while a service is executing. The default value is <code>false</code>. You cannot control <code>showBusyCursor</code> for individual operations or methods. As a workaround, you can use different services for the same WSDL document or class.</p> |
| <code>useProxy</code> | <p>For web services and HTTP services, you can bypass the Flex proxy by setting the <code>useProxy</code> property to <code>false</code> in an <code><mx:WebService></code> tag or <code><mx:HTTPService></code> tag, depending on the value of the corresponding <code><proxy-use-policy></code> tag in the <code><web-service-proxy></code> tag or <code><http-service-proxy></code> tag in the <code>flex-config.xml</code> file.</p> |

Remote-object-specific properties

The `<mx:RemoteObject>` tag contains the following properties in addition to those described in [“Common data service properties” on page 697](#):

| Property | Description |
|-----------------------|--|
| <code>endpoint</code> | <p>A specific alternative gateway endpoint. This property overrides the default (global) gateway endpoint in the <code><amf-gateway></code> tag of the <code><remote-objects></code> tag in the <code>flex-config.xml</code> file.</p> <p>You cannot use the <code>type</code> property if you change the endpoint to a non-Flex AMF gateway.</p> <p>The <code>endpoint</code> value can be a fully qualified path or a relative path. For a path that starts with <code>@ContextRoot()</code>, the endpoint is relative to the web application’s context root. For a path that starts with a slash (/), the endpoint is prepended to the protocol, host, and port of the Flex application.</p> |
| <code>id</code> | The instance name that you use to refer to the Java object in ActionScript code. |
| <code>named</code> | A named service that is specified in the <code>flex-config.xml</code> file. If you use the <code>named</code> property, you cannot use the <code>source</code> property. |
| <code>source</code> | Required for unnamed services; the source of the object. |
| <code>type</code> | <p>The type of object access. The default type is <code>stateless-class</code>. The following values are permitted:</p> <ul style="list-style-type: none"><code>stateless-class</code> A plain old Java object (POJO) or JavaBean; each method call creates a new object instance. This is the default value.<code>stateful-class</code> A POJO or JavaBean; the object state is maintained and each method call is made on the same object instance.<code>servlet</code> A Java servlet. The <code>servlet</code> type is used with the session servlet; for more information, see “Using a service with binding, validation, and event handlers” on page 669. <p>The <code>type</code> property is only supported for the Flex AMF gateway, and cannot be used for legacy endpoints (gateways).</p> <p>For named services, you can specify <code>type</code> in the <code>flex-config.xml</code> file. If you specify <code>type</code> in both places, the values must match or an error is reported.</p> |

Web-service-specific properties

The `<mx:WebService>` tag contains the following properties in addition to those described in [“Common data service properties” on page 697](#):

| Property | Description |
|----------------------|--|
| <code>load</code> | The value of the event handler for successful web service initialization. |
| <code>service</code> | A specific service name. Use when a WSDL document contains more than one service definition. |

| Property | Description |
|-------------|---|
| port | <p>A specific port name. Use when a WSDL document contains more than one port definition. You only need to use a <code>port</code> property when a WSDL <code><service></code> tag contains more than one <code><port></code> tag that uses SOAP. Many web services contain ports that use HTTP Get and HTTP Post in addition to SOAP. For example, you would not need to specify a port for the following service because only one port uses SOAP:</p> <pre> <service name="Shakespeare"> <port name="ShakespeareSoap" binding="s0:ShakespeareSoap"> <soap:address location="http://www.xmlme.com/WSShakespeare.asmx" /> </port> <port name="ShakespeareHttpGet" binding="s0:ShakespeareHttpGet"> <http:address location="http://www.xmlme.com/WSShakespeare.asmx" /> </port> <port name="ShakespeareHttpPost" binding="s0:ShakespeareHttpPost"> <http:address location="http://www.xmlme.com/WSShakespeare.asmx" /> </port> </service> </pre> |
| serviceName | A named service that is specified in the flex-config.xml file. If you use the <code>serviceName</code> property, you cannot use the <code>wsdl</code> property . |
| wsdl | The WSDL document for a web service. |

Note: The capitalization of WSDL URLs and the names of operations, arguments, services, and ports must match those that you defined for the web service.

HTTP-service-specific properties

The `<mx:HTTPService>` tag contains the following properties in addition to those described in [“Common data service properties” on page 697](#):

| Property | Description |
|--------------|---|
| contentType | <p>The type of content for service requests. The following values are permitted:</p> <ul style="list-style-type: none"> <code>application/x-www-form-urlencoded</code> (default) Sends requests like a normal HTTP POST with name-value pairs. <code>application/xml</code> Send requests as XML. |
| method | The HTTP method for sending the request. Permitted values are <code>GET</code> and <code>POST</code> ; lowercase values are converted to uppercase. The default value is <code>GET</code> . |
| resultFormat | <p>The value that indicates how you want to deserialize the result returned by the HTTP call. The value for this is based on the following:</p> <ul style="list-style-type: none"> Whether you are returning XML or name-value pairs. How you want to access the results; you can access results as an object, text, or XML. <p>The following values are permitted:</p> <ul style="list-style-type: none"> <code>object</code> The value returned is XML and is parsed as a tree of <code>ActionScript</code> objects. This is the default value. <code>xml</code> The value returned is XML, and it is returned as literal XML in an <code>ActionScript XMLNode</code> object. <code>flashVars</code> The value returned is text that contains name-value pairs, which is parsed into an <code>ActionScript</code> object. <code>text</code> The value returned is text, and it is left raw. |

| Property | Description |
|--------------------------|--|
| <code>serviceName</code> | A named service that is specified in the <code>flex-config.xml</code> file. If you use the <code>serviceName</code> property, you cannot use the <code>url</code> property. |
| <code>url</code> | The location of the service. The specified URL should not contain any query arguments (question mark character (?) followed by name-value pairs). |
| <code>xmlDecode</code> | <p>ActionScript function used to decode a service result from XML.</p> <p>When <code>resultFormat</code> is <code>object</code> and if the <code>xmlDecode</code> property is set, Flex uses the XML that the <code>HTTPService</code> object returns to create an object as specified in the <code>xmlDecode</code> function. If it is not defined, Flex uses the default <code>XMLDecoder</code> to create an object.</p> <p>The <code>xmlDecode</code> property takes an <code>XMLNode</code> object and should return an object. It can return any type of object, but it must return something. Returning null or undefined causes a fault.</p> |

The following example shows an `<mx:HTTPService>` tag that specifies an `xmlDecode` function:

```
<mx:HTTPService id="hs" xmlDecode="xmlDecoder" url="myURL"
    resultFormat="object" contentType="application/xml">
    <mx:request><source/>
        <obj>{RequestObject}</obj>
    </mx:request>
</mx:HTTPService>
```

The following example shows an `xmlDecoder` function:

```
function xmlDecoder ( myXML ) {

    // Simplified decoding logic.

    var myObj = new Object();

    myObj.name = myXML.firstChild.nodeValue;

    myObj.honorific = myXML.firstChild.attributes.honorific;

    return myObj;

}
```

| Property | Description |
|-----------|--|
| xmlEncode | <p>ActionScript function used to encode a service request as XML.</p> <p>When the contentType of a request is application/xml and the request object passed in is an object, Flex attempts to use the function specified in the xmlEncode property to turn it into XML. If the xmlEncode property is not set, Flex uses the default XMLEncoder to turn the object graph into XML.</p> <p>The xmlEncode property takes an object (you may have specified <code>httpService.request = new MyClass()</code>) and should return an XMLNode. In this example, the XMLNode object can be an XML object, which is a subclass of XMLNode, or the first child of the XML object, which is what you get from an <code><mx:XML></code> tag. Returning the wrong type of object causes a fault.</p> <p>The following example shows an <code><mx:HTTPService></code> tag that specifies an xmlEncode function:</p> <pre> <mx:HTTPService id="hs" xmlEncode="xmlEncoder" url="myURL" resultFormat="object" contentType="application/xml"> <mx:request><source/> <obj>{RequestObject}</obj> </mx:request> </mx:HTTPService> </pre> <p>The following example shows an xmlEncoder function:</p> <pre> function xmlEncoder (myObj) { return new XML("<userencoded><attrib0>MyObj.test</attrib0> <attrib1>MyObj.anotherTest</attrib1></userencoded>"); } </pre> |

Data service whitelist tags

This section describes the child tags that are common to the `<whitelist>` tags in the flex-config.xml file for remote object services, web services, and HTTP services, and the child tags that are specific to each type of service.

For information about whitelists, see [“Named services” on page 658](#) and [“Securing data services” on page 687](#).

Common whitelist tags

All `<service>` and `<object>` tags can contain the following child tags:

| Tag | Description |
|--|---|
| <code><allow-unnamed-access></code> | Adds the service to the unnamed whitelist. If you set this property to <code>false</code> , you cannot access the service using the <code>source</code> , <code>wsdl</code> , or <code>url</code> property of the <code><mx:RemoteObject></code> , <code><mx:WebService></code> , or <code><mx:HTTPService></code> tag, respectively. |
| <code><use-custom-authentication></code> | (Optional) Specifies that the service uses custom authentication instead of basic authentication. Use this when you provide a user name and password from the Flex application instead of a web browser dialog box. |

Remote-object-specific whitelist tags

Remote object service `<object>` tags can contain the following child tags in addition to those described in [“Common whitelist tags” on page 701](#):

| Tag | Description |
|---|--|
| <code><source></code> | The Java class to use for the service. |
| <code><type></code> | The type of class. Can be <code>stateless-class</code> , or <code>stateful-class</code> . The default type is <code>stateless-class</code> . You can also specify type in the <code><mx:RemoteObject></code> tag. If you specify type in both the <code><object></code> tag and the <code><mx:RemoteObject></code> tag, the values must match. |
| <code><use-basic-authentication></code> | Indicates whether the service uses Basic authentication. For more information, see “Securing data services” on page 687 . |

Web-service-specific whitelist tags

Web service `<service>` tags can contain the following child tags in addition to those described in [“Common whitelist tags” on page 701](#):

| Tag | Description |
|-------------------------------|---|
| <code><endpoint></code> | Endpoint URLs allowed when using the service. |
| <code><run-as></code> | The user name and password to use when accessing the service. |
| <code><wsdl></code> | WSDL URL for the service. |

HTTP-service-specific whitelist tags

HTTP service `<service>` tags can contain the following child tags in addition to those described in [“Common whitelist tags” on page 701](#):

| Tag | Description |
|-----------------------------|---|
| <code><url></code> | The service URL. |
| <code><run-as></code> | The user name and password to use when accessing the service. |

CHAPTER 31

Validating Data in Flex

This chapter describes the Macromedia Flex data validation feature. Data validators let you validate the fields of an object. For more information about data models, see [Chapter 29](#), “Binding and Storing Data in Flex,” on page 637.

Contents

| | |
|---|-----|
| Validating data | 703 |
| Using standard validators | 712 |

Validating data

The data that a user enters in a user interface might or might not be appropriate to the application. In Flex, you use a *validator* to ensure the values in the fields of an object meet certain criteria. For example, you can use a validator to ensure that a user enters a valid phone number value in a TextInput control. You can assign each field of an object to a single validator. The `mx.validators.Validator` class is an ActionScript class that you can extend to add your own validation logic.

You can call validation logic through ActionScript or by using an MXML tag. Flex includes a set of `mx.validators.Validator` subclasses for common types of user input data, such as ZIP codes, phone numbers, and credit cards. Validator subclass tags have one required property, the field to validate. The `<mx:Validator>` tag and Validator subclass tags must always be immediate children of the root tag of an MXML file.

You declare a validator in MXML using the `<mx:Validator>` tag or the tag for the appropriate Validator subclass. For example, to declare the standard `PhoneNumberValidator` validator, you use the `<mx:PhoneNumberValidator>` tag. To validate data that is bound to a data model, you should declare the appropriate validator in the same file as the bindings you make to the data model. If the validator and bindings aren't in the same file, the validator is not triggered when data is copied into the model; however, you could trigger validation in an ActionScript function. You do not have to declare the validator in the same file as the data model. For more information, see “[Triggering validation programmatically](#)” on page 704.

The following example shows an `<mx:PhoneNumberValidator>` tag that validates data entered in the *phone* field of a data model called *person*. The validation occurs when an event triggers a binding of data into the data model, or when you trigger validation in `ActionScript`.

```
<mx:PhoneNumberValidator field="person.phone" />
```

Validating multiple fields with one validator

The `<mx:PhoneNumberValidator>` tag validates a single data model field, but a validator can validate more than one field.

For example, you could create a custom validator called `NameValidator` to validate three data model fields that represent a person's first, middle, and last names. In dot notation, these fields would be represented as `person.name.first`, `person.name.middle`, and `person.name.last`.

To declare a validator called `NameValidator`, you use the following tag in MXML; because you are interested in all three children of the name field, you specify the parent field, `person.name`, in the `field` property:

```
<NameValidator field="person.name" />
```

The corresponding `NameValidator` class might look like the following class:

```
class NameValidator extends mx.validators.Validator
{
    public function doValidation(value) : Void {
        if (value.first == "") {
            validationError("noFirstName", "No First Name", "first");
        }
        if (value.middle == "") {
            validationError("noMiddleName", "No Middle Name", "middle");
        }
        if (value.last == "") {
            validationError("noLastName", "No Last Name", "last");
        }
    }
}
```

The three parameters of the `validationError()` method are `errorCode`, `defaultString`, and `subField`. For a custom validator on a single field, you do not need the third parameter, `subField`. If you do not set the `subField` parameter, the default value is `null`.

For more information about using custom `ActionScript` components, see [Chapter 14, “Creating ActionScript Components,”](#) on page 359.

Triggering validation programmatically

Validation is triggered automatically when a data binding executes, as the following example shows:

```
...
<mx:Model id="date">
    <month>{monthInput.text}</month>
    <day>{dayInput.text}</day>
    <year>{yearInput.text}</year>
</mx:Model>

<mx:TextInput id="monthInput"/>
```



```

<mx:TextInput id="dayInput"/>
<mx:TextInput id="yearInput"/>

<mx:DateValidator field="date"/>
...

```

Automatic validation only works when binding and validators are defined at the same level in the application. They must be defined in the same file.

You can also trigger validation programmatically in ActionScript. This is particularly useful when you want to validate something that cannot be set directly using binding. To trigger validation in ActionScript, you call the following static method of the `Validator` class either in your own function or as part of an event handler property:

```
mx.validators.Validator.isValid(objectContainingField, "field")
```

The *objectContainingField* parameter is usually the MXML document, which is represented as the `this` keyword. The `"field"` parameter is a String that represents the ID of the field to validate; this is often a nested property that requires dot notation. To determine the appropriate validator, Flex matches the *field* value in the `isValid()` method to the `field` value of a validator. For example, you use the code to execute the `PhoneNumberValidator` for the `person.phone` field:

```
mx.validators.Validator.isValid(this, "person.phone");
```

The `mx.validators.Validator` class also contains another static method called `isStructureValid()` that calls all validators assigned to the fields of an object. This is particularly useful for validating a form that has fields bound to a data model. For more information about the `isStructureValid()` method, see [“Validating a form” on page 707](#).

Working with validation events and error messages

When validation fails, a validator raises a `validationFailed` event intended for objects that are registered to be notified that a field is invalid. A validator raises a `validationSucceeded` event when a field is valid. UIComponent subclasses, which include the majority of Flex components, generally handle events by changing border color, displaying an error message, or hiding an error message.

Validators generate error messages as part of the `validationFailed` event. For example, a `PhoneNumberValidator` might contain an error message that indicates a number has the wrong number of digits. You can override an error message by assigning a new one as a property of the validator tag; for example:

```

<mx:PhoneNumberValidator field="person.phone" wrongNumError="Phone numbers
  have 7 or 10 digits" />

```

You can use the same technique to set any configuration parameters that a validator requires.

All validator tags support a `listener` property that points to an object that can handle `validationFailed` and `validationSucceeded` events. All `UIComponents` can handle these events. The `listener` property is optional. If you do not specify a listener, the compiler attempts to determine if the component that is the source of a data binding is an appropriate listener. If the compiler does not find an appropriate listener, errors occur in the `Application` object at the top level of the application. In the following validator tag, a `Form` container called `personForm` is assigned as the event listener:

```
<NameValidator field="person.name" listener="personForm" />
```

The listener is invoked after a validator has finished gathering all its errors. The `validationSucceeded` event has a `field` property, and the `validationFailed` event has `field` and `errors` properties. The `errors` property is an `Array` of error messages. The `message` property of the `validationFailed` event consolidates all `errors` property messages into one string.

Validating data in a custom validation function

You can place validation logic in a function in an `<mx:Script>` tag instead of in a validator class when you validate data in a limited scope, or you make one function call to invoke validation logic on a set of validator classes. You assign the validation logic contained in the function by placing a code snippet in the `<mx:Validator>` tag's `validate` property.

The `<mx:Validator>` tag in the following example calls the `myValidateName()` function contained in the `<mx:Script>` tag. This is a simple validation that validates a person's first, middle, and last names. The event is passed to the code snippet where `validator` is the `Validator` instance and `value` is the value to validate, which in this example is *person.name*. The validator's `validationError()` method takes three arguments that specify the error name, the error message, and the subfield of the value to which the error message applies.

```
<mx:Validator field="person.name"
  validate="myValidateName(event.validator, event.value)" />
<mx:Script>
  <![CDATA[

    function myValidateName(validator, value) {
      if (value.first == "") {
        validator.validationError("noFirstName", "First name is required",
          "first");
      }
      if (value.middle == "") {
        validator.validationError("noMiddleName", "Middle name is
          required", "middle");
      }
      if (value.last == "") {
        validator.validationError("noLastName", "Last name is required",
          "last");
      }
    }

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

Validating a form

Flex provides two very useful ways to validate the multiple fields of data that make up a form. One way to validate multiple fields is by using the static validation methods of the standard validators; these methods let you reuse and extend the validators in another validator class or validator function. Another way to validate multiple fields is by using the validator's `isStructureValid()` method. This static method lets you validate all or part of an object with one method call when there are validators assigned to the object's fields.

Calling standard validators from another validator

You can use a custom validator function as a single validator that calls standard validators. The `customValidate()` function in the following example calls the `ZipCodeValidator` and the `PhoneNumberValidator` to validate the `zipCode` and `phoneNumber` fields of the `formmodel` data model.

```
<!-- Here is the data model. -->
<mx:Model id="formmodel">
    <zipCode>{zip.text}</zipCode>
    <phoneNumber>{phone.text}</phoneNumber>
</mx:Model>

<!-- Here is the validator tag. -->
<mx:Validator field="formmodel" validate="customValidate(event.validator,
    event.value)" listener="this" />

<!-- Here is the validation function that uses both the ZipCodeValidator and
    PhoneNumberValidator to validate the form. -->
<mx:Script>
< ![[CDATA[
    function customValidate(validator, value) : Void {
        mx.validators.ZipCodeValidator.validateZipCode(validator,
            value.zipCode, null, "zipCode");
        mx.validators.PhoneNumberValidator.validatePhoneNumber(validator,
            value.phoneNumber, null, "phoneNumber");

        // If either of the fields validated above is invalid, don't move on
        // to the next field.
        if (validator.hasErrors()) return;

        // Now check if the phone number and ZIP code work together.
        // Use areaCodeMap, which was declared elsewhere.
        if
            (!areaCodeMap.get(parseAreaCode(value.phoneNumber)).
                contains(value.zipCode))
        {
            // This is a form-level error, so there is no subfield.
            validator.validationError("areaZipMismatch", "The zip code cannot
                have this phone number", null);
        }
    }
]]>
</mx:Script>
```

In the previous example, you trigger validation programmatically, as described in [“Triggering validation programmatically” on page 704](#). In the following example, validation is triggered in the `click` property of a `Button` control:

```
<mx:Button click="mx.validators.Validator.isValid(this, 'formmodel');  
    MyStockService.GetQuotes.send();" />
```

The `customValidate()` function in the previous example calls two static convenience methods, the `validateZipCode()` and `validatePhoneNumber()` methods of the standard `ZipCodeValidator` and `PhoneNumberValidator` validators. These static methods validate the `zipCode` and `phoneNumber` fields of the `formmodel` data model. These methods, and corresponding methods on the other standard validators, take the following four arguments:

- Argument one, `validator`, is the `Validator` instance.
- Argument two is `value.subfield`, where *subfield* is a subfield of the value specified in the `field` property of the `<mx:Validator>` tag.
- Argument three is an object containing parameter information, such as error messages or minimum to maximum values. This argument is optional, but if you specify argument four, you should pass `null`.
- Argument four is a text representation of the subfield specified in argument two. For example, if argument two is `value.zipCode`, argument four is `"zipCode"`. If argument two is `value.creditCard.cardType`, argument three is `"creditCard.cardType"`. This is to assist in error message creation.

The three parameters of the `validationError()` method are `errorCode`, `defaultString`, and `subField`. For a custom validator on a single field, you do not need the third parameter, `subField`. If you do not set the `subField` parameter, the default value is `null`.

When you call multiple validators at the same time, as in the `customValidate()` function, the errors accumulate. When a validation error is detected, you can call the `validator.hasErrors()` method and return the errors to stop processing. The `customValidate()` function calls `validator.hasErrors()` after validating the `zipCode` and `phoneNumber` fields.

The `customValidate()` function also contains a custom validation error specified in a call to the `validator.validationError()` method. The first and second arguments of the method specify the error code and error message, respectively. The third argument indicates the subfield relative to the object being validated that caused the error. When the validator is executing, it knows its own assigned field and all validation errors are based on that string. If there is a listener on the validator for the field, the error goes to that listener. If you pass `null` as the third argument of the `validationError()` method, as in the `customValidate()` function, the function uses the validator's field. The listener for the `formmodel` data model is set to `this`, which is the application object. If you specify a subfield in the third argument, the function uses the validator's field plus whatever you pass in.

In the static validation methods, you pass in the validator that knows its own base. Then you pass in a `baseField`, which is appended to that base before the static validator method adds its own subfield.

For more information about the standard validators included with Flex, see [“Using standard validators” on page 712](#).

Using the `isStructureValid` method to validate an object

The `Validator.isStructureValid()` method is very useful for validating the fields of a form that are bound to a particular object. You can use this method to call all validators assigned to the fields of a specified object.

One way to use the `isStructureValid()` method is to declare an `<mx:Validator>` tag that specifies the field to validate and uses the `isStructureValid()` method as the value of its `validate` property. For example, the `<mx:Validator>` tag in the following code validates the `foo` data model and runs the individual validators assigned to `foo`'s fields:

```
<mx:Model id="foo">
  <bar>{inp1.text}</bar>
  <baz>{inp2.text}</baz>
</mx:Model>

<mx:ZipCodeValidator field="foo.bar" />
<mx:PhoneNumberValidator field="foo.baz" />
<mx:Validator field="foo" validate="Validator.isStructureValid(this, 'foo');"
/>
<mx:Button id="myButton" click="mx.validators.Validator.isStructureValid(this,
'foo');"/>
```

For this example, the `ZipCodeValidator` and `PhoneNumberValidator` validators could be called in any of the following ways:

- Whenever the bindings in the `<bar>` and `<baz>` elements execute, the validators execute.
- When `myButton` is clicked and the `isStructureValid()` method is called on the `foo` data model, the validators execute.
- If the `isValid()` method is called on the `foo` data model, it calls the `isStructureValid()` method, and the validators execute.

The `isStructureValid()` method is also called automatically before sending requests to web services and Java objects. Any validators assigned to parameters of web service requests or arguments of Java object methods are automatically called.

In the following example, the `ZipCodeValidator` validator assigned to a web service request parameter and a Java object (`RemoteObject`) method argument are called before the requests are sent:

```
<!-- Web service object handles web service requests and results
(the specified WSDL URL is not functional). -->
<mx:WebService id="WeatherService" wsdl="/ws/WeatherService?wsdl">
  <mx:operation name="GetWeather"
    <mx:request>
      <ZipCode>{myZipField.text}</ZipCode>
    </mx:request>
  </mx:operation>
</mx:WebService>

<mx:ZipCodeValidator field="WeatherService.GetWeather.request.ZipCode"/>

<mx:RemoteObject id="Weather" src="weatherpackage.Weather">
  <mx:method name="getWeather">
```

```

        <mx:arguments>
            <zipCode>{inp.text}</zipCode>
        </mx:arguments>
    </mx:method>
</mx:RemoteObject>

<mx:ZipCodeValidator field="Weather.getWeather.arguments.zipCode" />

```

Another way to use the `isStructureValid()` method is from within a function that is called just before a web service request or Java object request is sent. The code in the following example calls the `isStructureValid()` method in the `validateWebService()` function. If errors are detected on any of the fields in the `myModel` data model, the request is not submitted to the web service, and the validation error message “There are invalid inputs to the web service.” is generated.

```

<mx:WebService id="WeatherService" ...>
    <mx:operation name="getTemp">
        <mx:request>
            <zipCode>{myModel.zipCode}</zipCode>
        </mx:request>
    </mx:operation>
</mx:WebService>

<mx:Model id="myModel">
    <zipCode>{zipInput.text}</zipCode>
</mx:Model>

<mx:ZipCodeValidator field="myModel.zipCode" />

<!--Link up the web service request validation to the model. -->
<mx:Validator field="WeatherService.getTemp.request"
    validate="validateWebService(event.validator);" />

<mx:TextInput id="zipInput" />

<mx:Script>
<![CDATA[
function validateWebService(validator) {
    if (!mx.validators.Validator.isStructureValid(this, "myModel")) {
        validator.validationError("modelInvalid", "There are invalid inputs to
        the web service.", null);
    }
}
]]>
</mx:Script>

```

Validating complex objects

When you validate an object that contains multiple properties that are set independently, there is no way to automatically determine when to trigger the validator because no field is directly bound to the object. For example, the `CreditCardValidator` takes a complex object that contains two properties, `cardType` and `cardNumber`, which are set independently.

One way to validate a complex object is by adding an event handler that triggers the validator based on some type of user interaction. In the following example, the `focusOut` event handler on the `cardNumber` `TextArea` control is a call to the `Validator.isValid()` method. The credit card type and credit card number are both validated when focus leaves the `cardNumber` `TextArea` control.

```
<mx:CreditCardValidator field="myModel.creditCard"/>
<mx:Model id="myModel">
    <creditCard>
        <cardType>{cardTypeRadio.selectedData}</cardType>
        <cardNumber>{cardNumber.text}</cardNumber>
    </creditCard>
</mx:Model>
<mx:Form width="100%">
    <mx:FormItem label="Credit Card" width="100%">
        <mx:RadioButtonGroup id="cardTypeRadio" />
        <mx:RadioButton id="radioVisa" label="Visa" groupName="cardTypeRadio"/>
        <mx:RadioButton id="radioMC" label="MasterCard"
            groupName="cardTypeRadio"/>
        <mx:TextInput id="cardNumber" width="100%"
            focusOut="mx.validators.Validator.isValid(this,
                'myModel.creditCard');"/>
    </mx:FormItem>
</mx:Form>
```

Disabling and enabling a validator

The `Validator.disable(document, objName)` method lets you disable a validator. The `Validator.enable(document, objName)` method lets you enable a validator. These methods are useful when you want to reset a field that is a source for binding, and you want to clear the validator so it starts over. The following example uses both of these methods to reset validation on the `text` property of a `TextInput` control:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" height="600">

    <mx:Model id="model">
        <zipCode>{input.text}</zipCode>
    </mx:Model>

    <mx:ZipCodeValidator field="model.zipCode" />
    <mx:TextInput id="input" />
    <mx:Button label="reset without" click="resetWithout()" />
    <mx:Button label="reset with" click="resetWith()" />

    <mx:Script>
        <![CDATA[
            import mx.validators.Validator;

            <!-- Validation will execute, marking the field invalid. -->
            function resetWithout()
            {
                input.text = '';
            }
        ]]>
    </mx:Script>
```

```

        <!-- Validation will clear. -->
        function resetWith()
        {
            Validator.disable(this, "model.zipCode");
            resetWithout();
            Validator.enable(this, "model.zipCode");
        }
    </mx:Script>
</mx:Application>

```

Using standard validators

Flex includes the `mx.validators.Validator` subclasses described in the following sections. You can use these validators for common types of data, including credit card numbers, dates, e-mail addresses, numbers, phone numbers, Social Security numbers, strings, and ZIP codes. This section describes these validators.

CurrencyValidator

The `CurrencyValidator` class checks that a string is a valid currency expression based on a set of parameters. The `<mx:CurrencyValidator>` tag has the following properties:

| Property | Description |
|---------------------------------|---|
| <code>field</code> | (Required) The field to be validated. Always starts at the application level. |
| <code>allowNegative</code> | Whether the currency expression can represent a negative value. The default value is <code>true</code> . |
| <code>alignSymbol</code> | On which side of the expression that the currency symbol occurs. The default value is <code>any</code> . |
| <code>currencySymbol</code> | The currency symbol to use. The default value is <code>\$</code> . |
| <code>decimalSeparator</code> | The character used to separate the whole number part from the fractional part of the currency value. Must be one character only. The default value is <code>."</code> . |
| <code>listener</code> | The validation listener. Usually a <code>UIComponent</code> object. |
| <code>precision</code> | The maximum number of digits allowed to follow the <code>decimalSeparator</code> . The default value is <code>2</code> . |
| <code>required</code> | Whether a field is required. The default value is <code>true</code> . |
| <code>requiredFieldError</code> | Error message displayed when a user does not complete a required field. The default is <code>"This field is required."</code> . |
| <code>thousandsSeparator</code> | The character used to group every three digits in the whole number part of the currency value. Must be one character only. The default value is <code>","</code> . |

The `CurrencyValidator` class contains a static method called `validateCurrency()` that you can use to validate a currency expression within a validation function that validates a whole form. For more information, see [“Validating a form” on page 707](#).

Example

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >

<!-- Example for US currency. -->
  <mx:Model id="US">
    <currency>{priceUS.text}</currency>
  </mx:Model>

  <CurrencyValidator field="US.currency" alignSymbol="left" />

  <mx:Label text="Enter a US-formatted price:" />
  <mx:TextInput id="priceUS" />

<!-- Example for European currency. -->
  <mx:Model id="European">
    <currency>{priceEU.text}</currency>
  </mx:Model>

  <CurrencyValidator field="European.currency" currencySymbol="€"
    alignSymbol="right" decimalSeparator="," thousandsSeparator="." />

  <mx:Label text="Enter a European-formatted price:" />
  <mx:TextInput id="priceEU" />
</mx:Application>
```

CreditCardValidator

The `CreditCardValidator` class validates that a credit card number is the correct length for the specified card type:

- Visa: 13 or 16 digits
- MasterCard: 16 digits
- Discover: 16 digits
- American Express: 15 digits
- DinersClub: 14 digits

The `<mx:CreditCardValidator>` tag has the following properties:

| Property | Description |
|-------------------------|---|
| field | (Required) The field to be validated. Always starts at the application level. |
| allowedFormatChars | The formatting characters allowed. The default value is “ - ”. |
| invalidCharError | An error message. The default is “Invalid characters in your credit card number. (Only enter numbers.)” |
| invalidFormatCharsError | An error message. The default is “The allowedFormatChars parameter is invalid. It cannot contain any digits.” |
| invalidNumberError | An error message. The default is “The credit card number is invalid.” |

| Property | Description |
|--------------------|---|
| listener | The validation listener. Usually a UIComponent. |
| noNumError | An error message. The default is “No credit card number specified.” |
| noTypeError | An error message. The default is “No credit card type specified or the type is not valid.” |
| required | Whether a field is required. The default value is true. |
| requiredFieldError | Error message displayed when a user does not complete a required field. The default is “This field is required.”. |
| wrongLengthError | An error message. The default is “Your credit card number contains the wrong number of digits.” |
| wrongTypeError | An error message. The default is “Incorrect card type specified.” |

You can indicate the type of credit card number to validate by assigning the following constants:

- `CreditCardValidator.kMasterCard`
- `CreditCardValidator.kVisa`
- `CreditCardValidator.kAmericanExpress`
- `CreditCardValidator.kDiscover`
- `CreditCardValidator.kDinersClub`

The `CreditCardValidator` class contains a static method called `validateCreditCard()` that you can use to validate a credit card within a validation function that validates a whole form. For more information, see [“Validating a form” on page 707](#).

Example

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:Model id="creditcard">
    <cardType>{cardTypeCombo.selectedItem.data}</cardType>
    <cardNumber>{cardNumberInput.text}</cardNumber>
  </mx:Model>

  <mx:Script>
    <![CDATA[
      function initCredit()
      {
        cardTypeCombo.addItem("American
          Express",mx.validators.CreditCardValidator.kAmericanExpress);
        cardTypeCombo.addItem("DinersClub",
          mx.validators.CreditCardValidator.kDinersClub);
        cardTypeCombo.addItem("Discover",mx.validators.
          CreditCardValidator.kDiscover);
        cardTypeCombo.addItem("MasterCard",
          mx.validators.CreditCardValidator.kMasterCard);
        cardTypeCombo.addItem("Visa",mx.validators.
          CreditCardValidator.kVisa);
      }
    ]]>
  </mx:Script>
</mx:Application>
```

```

    ]]>
</mx:Script>

<mx:CreditCardValidator field="creditcard" listener="this"/>

<mx:Form id="creditCardForm">
  <mx:FormItem label="Card Type">
    <mx:ComboBox id="cardTypeCombo" initialize="initCredit()"/>
  </mx:FormItem>
  <mx:FormItem label="Credit Card Number">
    <mx:TextInput id="cardNumberInput"/>
  </mx:FormItem>
  <mx:FormItem>
    <mx:Button label="Check Credit"
      click="mx.validators.Validator.isValid(this,'creditcard');"/>
  </mx:FormItem>
</mx:Form>
</mx:Application>

```

DateValidator

The `DateValidator` subclass validates that a string or object is a proper date and matches a specified format. Users can enter a single digit or two digits for month and day. The

`<mx:DateValidator>` tag has the following properties:

| Property | Description |
|--------------------------------------|---|
| <code>field</code> | (Required) The field to be validated. Always starts at the application level. |
| <code>allowedFormatChars</code> | The formatting characters allowed for separating the month, day, and year values. The default value is <code>"/\-. "</code> . |
| <code>formatError</code> | An error message. The default is <code>"Configuration error: Incorrect formatting string."</code> |
| <code>inputFormat</code> | The date format to validate the value against. The default is <code>"mm/dd/yyyy"</code> ; <code>"mm"</code> is the month, <code>"dd"</code> is the day, and <code>"yyyy"</code> is the year. This string is case-sensitive. |
| <code>invalidCharError</code> | An error message. The default is <code>"Invalid characters in your date."</code> |
| <code>invalidFormatCharsError</code> | An error message. The default is <code>"The allowedFormatChars parameter is invalid. It cannot contain any digits."</code> |
| <code>listener</code> | The validation listener. Usually a <code>UIComponent</code> . |
| <code>required</code> | Whether a field is required. The default value is <code>true</code> . |
| <code>requiredFieldError</code> | Error message displayed when a user does not complete a required field. The default is <code>"This field is required."</code> |
| <code>validateAsString</code> | A Boolean value whose default value is <code>true</code> . If you set the value to <code>true</code> , it evaluates the value as a string, unless the value has a month, day, or year property. In most cases, you do not need to set this parameter. |
| <code>wrongDayError</code> | An error message. The default is <code>"Please enter a valid day for the month."</code> |

| Property | Description |
|------------------|---|
| wrongLengthError | An error message. The default is "Please type the date in the format <i>ValidInputFormat</i> ." |
| wrongMonthError | An error message. The default is "Please enter a month between 1 and 12." |
| wrongYearError | An error message. The default is "Please enter a year between 0 and 9999." |

The `DateValidator` class contains a static method called `validateDate()` that you can use to validate a date within a validation function that validates a whole form. For more information, see [“Validating a form” on page 707](#).

Example

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:Model id="date">
    <month>{monthInput.text}</month>
    <day>{dayInput.text}</day>
    <year>{yearInput.text}</year>
  </mx:Model>

  <mx:DateValidator field="date" listener="this"/>

  <mx:Form width="140" >
    <mx:FormItem label="Month">
      <mx:TextInput id="monthInput"/>
    </mx:FormItem>
    <mx:FormItem label="Day">
      <mx:TextInput id="dayInput"/>
    </mx:FormItem>
    <mx:FormItem label="Year">
      <mx:TextInput id="yearInput"/>
    </mx:FormItem>
    <mx:Button label="Check Date"
      click="mx.validators.Validator.isValid(this,'date');"/>
  </mx:Form>

  <!-- Alternate method -->

  <mx:Model id="alternateDate">
    <date>{dateInput.text}</date>
  </mx:Model>

  <mx:Form id="dateForm">
    <mx:FormItem id="dateItem" label="Date of Birth (dd*mm*yyyy)">
      <mx:TextInput id="dateInput"/>
    </mx:FormItem>
  </mx:Form>

  <mx:DateValidator field="alternateDate.date" inputFormat="dd/mm/yyyy"
    allowedFormatChars="*#~"/>
</mx:Application>
```

EmailValidator

The `EmailValidator` class validates that a string has an at sign character (@) and a period character (.) in the domain. You can use IP domain names if they are enclosed in square brackets; for example, `myname@[206.132.22.1]`. You can use individual IP numbers from 0 to 255.

The `<mx:EmailValidator>` tag has the following properties:

| Property | Description |
|--|--|
| <code>field</code> | (Required) The field to be validated. Always starts at the application level. |
| <code>invalidCharError</code> | An error message. The default is "Invalid characters in your email address." |
| <code>invalidDomainError</code> | An error message. The default is "The domain in your email address is incorrectly formatted." |
| <code>invalidIPDomainError</code> | An error message. The default is "The IP domain in your email address is incorrectly formatted." |
| <code>invalidPeriodsInDomainError</code> | An error message. The default is "The domain in your email address has continuous periods." |
| <code>listener</code> | The validation listener. Usually a <code>UIComponent</code> . |
| <code>missingAtSignError</code> | An error message. The default is "Missing an @ character in your email address." |
| <code>missingPeriodInDomainError</code> | An error message. The default is "The domain in your email address is missing a period." |
| <code>missingUsernameError</code> | An error message. The default is "The username in your email address is missing." |
| <code>required</code> | Whether a field is required. The default value is <code>true</code> . |
| <code>requiredFieldError</code> | Error message displayed when a user does not complete a required field. The default is "This field is required." |

The `EmailValidator` class contains a static method called `validateEmail()` that you can use to validate an e-mail address within a validation function that validates a whole form. For more information, see [“Validating a form” on page 707](#).

Example

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:Model id="contact">
    <homePhone>{homePhoneInput.text}</homePhone>
    <cellPhone>{cellPhoneInput.text}</cellPhone>
    <email>{emailInput.text}</email>
  </mx:Model>

  <mx:Form id="contactForm">
    <mx:FormItem id="homePhoneItem" label="Home Phone">
      <mx:TextInput id="homePhoneInput"/>
    </mx:FormItem>
```

```

    <mx:FormItem id="cellPhoneItem" label="Cell Phone">
      <mx:TextInput id="cellPhoneInput"/>
    </mx:FormItem>
    <mx:FormItem id="emailItem" label="Email">
      <mx:TextInput id="emailInput"/>
    </mx:FormItem>
  </mx:Form>

  <mx:PhoneNumberValidator field="contact.homePhone"
    listener="homePhoneInput"/>
  <mx:PhoneNumberValidator field="contact.cellPhone"
    listener="cellPhoneInput"/>
  <mx:EmailValidator field="contact.email" listener="emailInput"/>
</mx:Application>

```

NumberValidator

The `NumberValidator` class validates that a string is a valid number between the minimum and maximum values, and can also check whether the value is an integer. The

`<mx:NumberValidator>` tag has the following properties:

| Property | Description |
|--------------------------------------|--|
| <code>field</code> | (Required) The field to be validated. Always starts at the application level. |
| <code>domain</code> | The type of number to be validated. Permitted values are <code>real</code> and <code>int</code> . The default value is <code>real</code> . |
| <code>exceedsMaxError</code> | An error message. The default is "This number exceeds the maximum allowed value." |
| <code>integerError</code> | An error message. The default is "This number must be an integer." |
| <code>invalidCharError</code> | An error message. The default is "The input contains invalid characters." |
| <code>invalidFormatCharsError</code> | An error message. The default is "One of the formatting parameters is invalid". |
| <code>listener</code> | The validation listener. Usually a <code>UIComponent</code> . |
| <code>lowerThanMinError</code> | An error message. The default is "This number is lower than the minimum allowed value." |
| <code>minValue</code> | The minimum value for a valid number. Not used by default. |
| <code>maxValue</code> | The maximum value for a valid number. Not used by default. |
| <code>required</code> | Whether a field is required. The default value is <code>true</code> . |
| <code>requiredFieldError</code> | Error message displayed when a user does not complete a required field. The default is "This field is required." |

The `NumberValidator` class contains a static method called `validateNumber()` that you can use to validate a number within a validation function that validates a whole form. For more information, see [“Validating a form” on page 707](#).

Example

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:Model id="product">
    <quantity>{quantityInput.text}</quantity>
  </mx:Model>

  <mx:Form id="productForm">
    <mx:FormItem id="quantityItem"
      label="Number of Widgets (max 10 per customer)">
      <mx:TextInput id="quantityInput"/>
    </mx:FormItem>
  </mx:Form>

  <mx:NumberValidator field="product.quantity" listener="quantityInput"
    minValue="1" maxValue="10" domain="int"/>
</mx:Application>
```

PhoneNumberValidator

The `PhoneNumberValidator` class validates that a string is a valid phone number. The `<mx:PhoneNumberValidator>` tag has the following properties:

| Property | Description |
|-------------------------|---|
| field | (Required) The field to be validated. Always starts at the application level. |
| allowedFormatChars | Formatting characters allowed. The default value is “()- .+”. |
| invalidCharError | An error message. The default is “Invalid characters in your phone number.” |
| invalidFormatCharsError | An error message. The default is “The allowedFormatChars parameter is invalid. It cannot contain any digits.” |
| listener | The validation listener. Usually a <code>UIComponent</code> . |
| required | Whether a field is required. The default value is <code>true</code> . |
| requiredFieldError | Error message displayed when a user does not complete a required field. The default is “This field is required.”. |
| wrongLengthError | An error message. The default is “Your telephone number must be at least 10 digits long.” |

The `PhoneNumberValidator` class contains a static method called `validatePhoneNumber()` that you can use to validate a phone number within a validation function that validates a whole form. For more information, see [“Validating a form” on page 707](#).

Example

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:Model id="contact">
    <homePhone>{homePhoneInput.text}</homePhone>
```

```

        <cellPhone>{cellPhoneInput.text}</cellPhone>
        <email>{emailInput.text}</email>
    </mx:Model>

    <mx:Form id="contactForm">
        <mx:FormItem id="homePhoneItem" label="Home Phone">
            <mx:TextInput id="homePhoneInput"/>
        </mx:FormItem>
        <mx:FormItem id="cellPhoneItem" label="Cell Phone">
            <mx:TextInput id="cellPhoneInput"/>
        </mx:FormItem>
        <mx:FormItem id="emailItem" label="Email">
            <mx:TextInput id="emailInput"/>
        </mx:FormItem>
    </mx:Form>

    <mx:PhoneNumberValidator field="contact.homePhone"
        listener="homePhoneInput"/>
    <mx:PhoneNumberValidator field="contact.cellPhone"
        listener="cellPhoneInput"/>
    <mx:EmailValidator field="contact.email" listener="emailInput"/>
</mx:Application>

```

SocialSecurityValidator

The `SocialSecurityValidator` class validates that a string is a valid United States Social Security number. The `<mx:SocialSecurityValidator>` tag has the following properties:

| Property | Description |
|--------------------------------------|---|
| <code>field</code> | (Required) The field to be validated. Always starts at the application level. |
| <code>listener</code> | The validation listener. Usually a <code>UIComponent</code> . |
| <code>allowedFormatChars</code> | The formatting characters allowed. The default value is “ - ”. |
| <code>invalidCharError</code> | An error message. The default is “Invalid characters in your Social Security number.” |
| <code>invalidFormatCharsError</code> | An error message. The default is “The allowedFormatChars parameter is invalid. It can not contain any digits.” |
| <code>wrongFormatError</code> | An error message. The default is “Social Security number must be 9 digits or in the form NNN-NN-NNNN.” |
| <code>zeroStartError</code> | An error message. The default is “Invalid SSN: SSN's can't start with 000.” |
| <code>required</code> | Whether a field is required. The default value is <code>true</code> . |
| <code>requiredFieldError</code> | Error message displayed when a user does not complete a required field. The default is “This field is required.”. |

The `SocialSecurityValidator` class contains a static method called `validateSocialSecurity()` that you can use to validate a Social Security number within a validation function that validates a whole form. For more information, see [“Validating a form” on page 707](#).

Example

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:Model id="identity">
    <socialSecurity>{ssnField.text}</socialSecurity>
    <driversLicense>{licenseInput.text}</driversLicense>
  </mx:Model>

  <mx:Form id="identityForm">
    <mx:FormItem id="ssnItem" label="Social Security Number">
      <mx:TextInput id="ssnField"/>
    </mx:FormItem>
    <mx:FormItem id="licenseItem" label="Driver's License Number">
      <mx:TextInput id="licenseInput"/> <!-- Not validated -->
    </mx:FormItem>
  </mx:Form>

  <mx:SocialSecurityValidator field="identity.socialSecurity"
    listener="ssnField"/>
</mx:Application>
```

StringValidator

The `StringValidator` class validates that a string length is within a specified range. The `<mx:StringValidator>` tag has the following properties:

| Property | Description |
|--------------------|--|
| field | (Required) The field to be validated. Always starts at the application level. |
| listener | The validation listener. Usually a <code>UIComponent</code> . |
| minLength | The minimum length for a valid string. Not used by default. |
| maxLength | The maximum length for a valid string. Not used by default. |
| required | Whether a field is required. The default value is <code>true</code> . |
| requiredFieldError | Error message displayed when a user does not complete a required field. The default is "This field is required." |
| tooLongError | An error message. The default is "This string is longer than the maximum allowed length." |
| tooShortError | An error message. The default is "This string is shorter than the minimum allowed length." |

The `StringValidator` class contains a static method called `validateString()` that you can use to validate a string within a validation function that validates a whole form. For more information, see [“Validating a form” on page 707](#).

Example

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:Model id="membership">
    <username>{userNameInput.text}</username>
    <fullName>{fullNameInput.text}</fullName>
  </mx:Model>

  <mx:Form id="membershipForm">
    <mx:FormItem id="fullNameItem" label="Full Name">
      <!-- Not validated -->
      <mx:TextInput id="fullNameInput"/>
    </mx:FormItem>
    <mx:FormItem id="userNameItem" label="Username">
      <mx:TextInput id="userNameInput"/>
    </mx:FormItem>
  </mx:Form>

  <mx:StringValidator field="membership.username" listener="userNameInput"
    minLength="6" maxLength="12"/>
</mx:Application>
```

ZipCodeValidator

The `ZipCodeValidator` class validates that a string has the correct length for a five-digit ZIP code or a five-digit+four-digit United States ZIP code or Canadian postal code. The

`<mx:ZipCodeValidator>` tag has the following properties:

| Property | Description |
|-------------------------|---|
| field | (Required) The field to be validated. Always starts at the application level. |
| allowedFormatChars | The formatting characters allowed. The default value is " -". |
| domain | The type of ZIP codes to check. Permitted values are <code>US Only</code> and <code>US or Canada</code> . The default value is <code>US Only</code> . |
| invalidCharError | An error message. The default is "Invalid characters in your zip code." |
| invalidDomainError | An error message. The default is "The domain parameter is invalid. It must be either 'US Only' or 'US or Canada'." |
| invalidFormatCharsError | An error message. The default is "The allowedFormatChars parameter is invalid. It cannot contain any digits or letters." |
| listener | The validation listener. Usually a <code>UIComponent</code> . |
| required | Whether a field is required. The default value is <code>true</code> . |
| requiredFieldError | Error message displayed when a user does not complete a required field. The default is "This field is required." |
| wrongCAFormatError | An error message. The default is "The Canadian zip code must be formatted like 'A1B 2C3'." |

| Property | Description |
|--------------------|---|
| wrongLengthError | An error message. The default is “Zip code must be 5 digits or 5+4 digits.” |
| wrongUSFormatError | An error message. The default is “The zip+4 extension must be formatted like '12345-6789'.” |

The `ZipCodeValidator` class contains a static method called `validateZipCode()` that you can use to validate a ZIP code within a validation function that validates a whole form. For more information, see [“Validating a form” on page 707](#).

Example

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:Model id="address">
    <zipCode>{zipInput.text}</zipCode>
  </mx:Model>

  <mx:Form id="addressForm">
    <mx:FormItem id="zipCodeItem" label="Zip Code">
      <mx:TextInput id="zipInput"/>
    </mx:FormItem>
  </mx:Form>

  <mx:ZipCodeValidator field="address.zipCode" domain="US or Canada"
    listener="zipInput"/>
</mx:Application>
```


CHAPTER 32

Formatting Data

This chapter describes how to use data formatters, user-configurable objects that format raw data into a customized string. You often use formatters with data binding to create a meaningful display of the raw data bound to a component. This can save you time by automating data formatting tasks and by letting you easily change the formatting of fields within your applications.

Contents

| | |
|---|-----|
| Using formatters | 725 |
| Writing an error handler function | 726 |
| Using the standard formatters | 726 |
| Creating a custom formatter | 738 |

Using formatters

Macromedia Flex formatters are ActionScript components that you use to format data into strings. Formatters perform a one-way conversion of raw data to a formatted string. They are triggered just before data is displayed in a text field. Flex includes standard formatters that let you format currency, dates, numbers, phone numbers, and ZIP codes.

All Flex formatters are subclasses of the `mx.formatters.Formatter` class. The `Formatter` class declares a `format()` method that takes a value and returns a `String`.

For most formatters, when an error occurs, an empty string is returned and a description is saved to an `error` property. The `error` property is inherited from the `Formatter` superclass.

The following procedure describes the general process for using a formatter:

1. Declare a formatter in your MXML code, specifying the appropriate formatting properties.
2. Call the formatter's `format()` method within the curly braces (`{ }`) syntax for binding data, and specify the value to be formatted as an argument to the `format()` method.

The following example declares a DateFormatter with an MM/DD/YYYY date format, and binds the formatted version of a Date object returned by a web service to the text property of a TextInput control:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
...
  <!-- Declare a formatter and specify formatting properties. -->
  <mx:DateFormatter id="StandardDateFormat" formatString="MM/DD/YYYY"/>

  <!-- Trigger the formatter while populating a string with data. -->
  <mx:TextInput text="Your order shipped on
    {StandardDateFormat.format(ws.result.date)}/>
</mx:Application>
```

Writing an error handler function

When you use a formatter, you can write an error handler function so that the user does not see error messages by default. You can also use an error function for debugging, but not in production because you should guarantee that valid values are passed to the formatter before an application goes into production. The following example shows a simple error handler function:

```
// This function would be in an MXML file.
function formatWithError(value) : String
{
    var formatted = myFormatter.format(value);
    if (formatted == "")
    {
        if (myFormatter.error != undefined)
        {
            if (myFormatter.error == "Invalid value")
            {
                formatted = "The value used in the format function is not a valid
                value.";
            }
            else
            {
                formatted = "The formatString provided is not a valid formatString.";
            }
        }
    }
    return formatted;
}
```

Using the standard formatters

This section describes the standard formatters included with Flex:

- NumberFormatter
- CurrencyFormatter
- PhoneFormatter
- ZipCodeFormatter
- DateFormatter
- SwitchSymbolFormatter

Using the NumberFormatter

The `NumberFormatter` class provides basic formatting options for numeric data, including decimal formatting, thousands separator formatting, and negative sign formatting. The `format()` method accepts a number or a number formatted as a `String`, and formats the resulting string.

When a number formatted as a `String` is passed to the `format()` method, the function parses the string from left to right and attempts to extract the first sequence of numbers it encounters. Thousands separators and decimal separators are included along with their trailing numbers. The parser searches for a comma (,) for the thousands separator unless a different character is set in the `thousandsSeparatorFrom` property. The parser searches for a period (.) for the decimal separator unless a different character is defined in the `decimalSeparator` property.

Note: The `format()` method recognizes a dash (-) immediately preceding the first number in the sequence as a negative number. A dash, space, and then number sequence is not interpreted as a negative number.

The following table describes the formatting options available as optional properties of the `<mx:NumberFormatter>` tag:

| Property | Type | Default value | Description |
|-------------------------------------|---------|---------------|--|
| <code>decimalSeparatorFrom</code> | String | . (period) | The user-definable separator character used when parsing a number string during input. |
| <code>decimalSeparatorTo</code> | String | . (period) | The user-definable character to use as a decimal separator when formatting the output string. |
| <code>precision</code> | Number | 2 | The number of decimal places to include in the formatting. If the precision value is set, numbers are all formatted at that precision. If the precision value is not set, a number's precision is unchanged during formatting. |
| <code>rounding</code> | String | none | The instructions for rounding the number. Permitted values are <code>none</code> , <code>up</code> , <code>down</code> , and <code>nearest</code> . |
| <code>thousandsSeparatorFrom</code> | String | , (comma) | The user-definable character to use as a thousands separator while parsing a number string during input. |
| <code>thousandsSeparatorTo</code> | String | , (comma) | The user-definable character to use as a thousands separator when formatting the output string. |
| <code>useNegativeSign</code> | Boolean | true | The negative number display value. If <code>true</code> , uses a negative sign for negative numbers. If <code>false</code> , renders a negative value in parentheses; for example, (400). |
| <code>useThousandsSeparator</code> | Boolean | true | The large number display value. If <code>true</code> , the <code>NumberFormatter</code> class uses a thousands separator to delimit every three digits. |

The rounding and precision values affect the formatting of the decimal in a number. If you use both rounding and precision properties, rounding is applied first, and then the decimal length is set using the specified precision value. This lets you round a number and still have a trailing decimal; for example, 303.99 = 304.00.

Example

The following example shows the `NumberFormatter` class in an MXML file:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">

    <!-- Declare and define parameters for the NumberFormatter. -->
    <mx:NumberFormatter id="PrepForDisplay"
        precision="0"
        rounding="up"
        decimalSeparatorTo="."
        thousandsSeparatorTo=","
        useThousandsSeparator="true"
        useNegativeSign="true" />

    <mx:Script>
        <![CDATA[
            var bigNumber = 6000000000.65;
        ]]>
    </mx:Script>
    <!-- Trigger the formatter while populating a string with data. -->
    <mx:TextInput text="{PrepForDisplay.format(bigNumber)}" />

</mx:Application>
```

At runtime, the following text appears:

6,000,000,001

Error handling

If an error occurs, Flex returns an empty string and saves a description of the error in the `error` property. An error refers to a problem with the value being submitted or the format string that contains the user settings, as described in the following table:

| Error type | When error occurs |
|----------------|--|
| Invalid value | An invalid numeric value is passed to the <code>format()</code> method. The value should be a valid number in the form of a <code>Number</code> or a <code>String</code> . |
| Invalid format | Any of the parameters contain an unusable setting. |

Using the `CurrencyFormatter`

The `CurrencyFormatter` class provides the same features as the `NumberFormatter` class, plus a currency symbol. It has two additional properties, `currencySymbol` and `alignSymbol`. For more information about the `NumberFormatter` class, see [“Using the NumberFormatter” on page 727](#).

The `CurrencyFormatter` class provides basic formatting options for numeric data, including decimal formatting, thousands separator formatting, and negative sign formatting. The `format()` method accepts a `Number` or a number formatted as a `String` and formats the resulting string.

When a number formatted as a `String` is passed to the `format()` method, the function parses the string from left to right and attempts to extract the first sequence of numbers it encounters. Thousands separators and decimal separators are included along with their trailing numbers. The parser searches for a comma (,) for the thousands separator unless a different character is set in the `thousandsSeparatorFrom` property. The parser searches for a period (.) for the decimal separator unless a different character is defined in the `decimalSeparator` property.

Note: When a number is provided to the `format()` method as a `String`, a negative sign is recognized if it is a dash (-) immediately preceding the first number in the sequence. A dash, space, and then a first number is not interpreted as a negative sign.

The following table describes the formatting options available as optional properties of the `<mx:CurrencyFormatter>` tag:

| Property | Type | Default value | Description |
|-------------------------------------|--------|-------------------------|---|
| <code>alignSymbol</code> | String | <code>left</code> | Aligns currency symbol to the left side or the right side of the formatted number. Permitted values are <code>left</code> and <code>right</code> . |
| <code>currencySymbol</code> | String | <code>\$</code> | User-definable character to use as a currency symbol for a formatted number. You can use one or more characters to represent the currency symbol; for example, <code>£</code> or <code>¥</code> . You can also use empty spaces to add space between the currency character and the formatted number. When the number is a negative value, the currency symbol appears between the number and the minus sign or parentheses. |
| <code>decimalSeparatorFrom</code> | String | <code>.</code> (period) | The user-definable separator character used when parsing a number string during input. |
| <code>decimalSeparatorTo</code> | String | <code>.</code> (period) | The user-definable separator character to use when outputting formatted decimal numbers. The <code>decimalSeparatorTo</code> value is also set when <code>precision</code> is adjusted. |
| <code>precision</code> | Number | <code>2</code> | The number of decimal places to include in the formatting. You can disable <code>precision</code> by setting it to <code>0</code> . |
| <code>rounding</code> | String | <code>none</code> | Rounds a number. Permitted values are <code>none</code> , <code>up</code> , <code>down</code> , and <code>nearest</code> . |
| <code>thousandsSeparatorFrom</code> | String | <code>,</code> (comma) | The user-definable character to use as a thousands separator while parsing a number string. |

| Property | Type | Default value | Description |
|-----------------------|---------|---------------|--|
| thousandsSeparatorTo | String | , (comma) | The user-definable character to use as a thousands separator when formatting the output string. |
| useNegativeSign | Boolean | true | If true, uses a negative sign for negative numbers. If false, renders a negative value in parenthesis; for example, (400). |
| useThousandsSeparator | Boolean | true | The useThousandsSeparator visually splits a number into thousands increments using a separator character. If true, the NumberFormatter class uses a thousands separator. |

Example

The following example shows the CurrencyFormatter class in an MXML file:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">

    <!-- Declare a CurrencyFormatter and define parameters. -->
    <mx:CurrencyFormatter id="Price" precision="2"
        rounding="none"
        decimalSeparatorTo="."
        thousandsSeparatorTo=","
        useThousandsSeparator="true"
        useNegativeSign="true"
        currencySymbol="$"
        alignSymbol="left" />
    <mx:Script>
        <![CDATA[
            var todaysPrice=4025;
        ]]>
    </mx:Script>

    <!-- Trigger the formatter while populating a string with data. -->
    <mx:TextInput text="Today's price is {Price.format(todaysPrice)}." />

</mx:Application>
```

At runtime, the following text is displayed:

Today's price is \$4,025.00.

Error handling

If an error occurs, an empty string is returned and a description of the error is saved to the `error` property. An error points to a problem with the value being submitted or the format string containing the user settings, as described in the following table:

| Error type | When error occurs |
|----------------|--|
| Invalid value | An invalid numeric value is passed to the <code>format()</code> method. The value should be a valid number in the form of a <code>Number</code> or a <code>String</code> . |
| Invalid format | Any of the parameters contain an unusable setting. |

Using the PhoneFormatter

The `PhoneFormatter` lets you format a phone number by adjusting the format of the area code and the subscriber code. You can also adjust the country code and configuration for international formats. The value passed into the `PhoneFormatter` must be a `Number` object or a `String` with only digits.

The `PhoneFormatter` `formatString` property accepts a formatted string as a definition of the format pattern. The following table shows common options for `formatString` values. The `PhoneFormatter`'s `format()` method accepts a sequence of numbers. The numbers correspond to the number of placeholder (`#`) symbols in the `formatString` value. The number of placeholder symbols in the `formatString` and the number of digits in the `format()` method value must match.

| formatString value | Input | Output |
|--------------------|---------------|-------------------|
| ###-#### | 1234567 | (xxx) 456-7890 |
| (###) ###-#### | 1234567890 | (123) 456-7890 |
| ###-###-#### | 11234567890 | 123-456-7890 |
| ##(###) ###-#### | 11234567890 | 1(123) 456 7890 |
| #-###-###-#### | 11234567890 | 1-123-456-7890 |
| +###-###-###-#### | 1231234567890 | +123-123-456-7890 |

In this table, dashes (-) are used as separator elements where applicable. You can substitute the dash characters with period (.) characters or blank spaces. You can change the default allowable character set as needed using the `validPatternChars` property. You can change the default character that represents a numeric placeholder by using the `numberSymbol` property (for example, to change from `#` to `$`).

Note: A shortcut is provided for the United States seven-digit format. If the `areaCode` property contains a value and you use the seven-digit format string, a seven-digit format entry automatically adds the area code to the string returned. The default format for the area code is `(###)`. You can change this using the `areaCodeFormat` property. You can format the area code any way you want as long as it contains three number placeholders.

The following table describes the formatting options available as optional properties of the `<mx:PhoneFormatter>` tag:

| Property | Type | Default value | Description |
|--------------------------------|--------|-----------------------------|---|
| <code>areaCode</code> | Number | No default | Area code number added to a seven-digit United States format phone number to form a ten-digit phone number. |
| <code>areaCodeFormat</code> | String | <code>(###)</code> | Default format for the area code when the <code>areaCode</code> property is rendered by a seven-digit format. |
| <code>formatString</code> | String | <code>(###) ###-####</code> | String containing mask characters representing a specified phone number format. |
| <code>numberSymbol</code> | String | <code>#</code> | Character to use as the number placeholder in the <code>formatString</code> property. |
| <code>validPatternChars</code> | String | <code>+,(),#,-...</code> | Comma-separated list of valid characters that you can use in the <code>formatString</code> property. This property is used during validation of the <code>formatString</code> . |

Example

The following example shows the `PhoneFormatter` class in an MXML file:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">

    <!-- Declare a PhoneFormatter and define formatting parameters. -->
    <mx:PhoneFormatter id="PhoneDisplay" areaCode="415" formatString="(###) ###-####"
        />

    <mx:Script>
        <![CDATA[
            var newNumber=1234567;
        ]]>
    </mx:Script>

    <!-- Trigger the formatter while populating a string with data -->
    <mx:TextInput text="{PhoneDisplay.format(newNumber)}" />

</mx:Application>
```

At runtime, the following text is displayed:

(415) 123-4567

Error handling

If an error occurs, an empty string is returned and a description of the error is saved to the `error` property. An error points to a problem with the value being submitted or the format string containing the user settings, as described in the following table:

| Error type | When error occurs |
|----------------|---|
| Invalid value | <ul style="list-style-type: none">An invalid numeric value is passed to the format method. The value should be a valid number in the form of a <code>Number</code> or a <code>String</code>.The value contains a different number of digits than what is specified in the format string. |
| Invalid format | <ul style="list-style-type: none">Any of the characters in the format string do not match the allowed characters specified in the <code>validPatternChars</code> property.The <code>areaCodeFormat</code> property is specified but does not contain exactly three numeric placeholders. |

Using the `ZipCodeFormatter`

The `ZipCodeFormatter` class lets you format five-digit or nine-digit United States ZIP codes and six-character Canadian postal codes. The `ZipCodeFormatter` class's `formatString` property accepts a formatted string as a definition of the format pattern. The `formatString` property is optional. If it is omitted, the default value of `#####` is used.

The number of digits in the value to be formatted and the value of the `formatString` property must be five or nine for United States ZIP codes, and six for Canadian postal codes.

The following table shows common `formatString` values, input values, and output values:

| <code>formatString</code> value | Input | Output | Format |
|---------------------------------|------------------|------------------------|------------------------------------|
| <code>#####</code> | 94117, 941171234 | 94117, 94117 | Five-digit U.S. ZIP code |
| <code>#####-####</code> | 941171234, 94117 | 94117-1234, 94117-0000 | Nine-digit U.S. ZIP code |
| <code>### ##</code> | A1B2C3 | A1B 2C3 | Six-character Canadian postal code |

For United States ZIP codes, if a nine-digit format is requested and a five-digit value is supplied, `-0000` is appended to the value to make it compliant with the nine-digit format. Inversely, if a nine-digit value is supplied for a five-digit format, the number is truncated to five digits.

For Canadian postal codes, only a six-digit value is allowed for either the `formatString` or the input value.

Note: For United States ZIP codes, only numeric characters are valid. For Canadian postal codes, alphanumeric characters are allowed. Alphabetic characters must be in uppercase.

Example

The following example shows the `ZipCodeFormatter` class in an MXML file:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">

    <!-- Declare a ZipCodeFormatter and define parameters. -->
```

```

<mx:ZipCodeFormatter id="ZipCodeDisplay" formatString="#####-####" />

<mx:Script>
  <![CDATA[
    var storedZipCode=123456789;
  ]]>
</mx:Script>

<!-- Trigger the formatter while populating a string with data -->
<mx:TextInput text="{ZipCodeDisplay.format(storedZipCode)}" />

</mx:Application>

```

At runtime, the following text is displayed:

12345-6789

Error handling

If an error occurs, an empty string is returned and a description of the error is saved to the `error` property. An error refers to a problem with the value being submitted or the format string containing the user settings, as described in the following table:

| Error type | When error occurs |
|----------------|---|
| Invalid value | <ul style="list-style-type: none"> An invalid numeric value is passed to the <code>format()</code> method. The value should be a valid number in the form of a <code>Number</code> or a <code>String</code>, except for Canadian postal code, which allows alphanumeric values. The number of digits does not match the allowed digits from the <code>formatString</code> property. |
| Invalid format | <ul style="list-style-type: none"> Any of the characters in the format string do not match the allowed characters specified in the <code>validFormatChars</code> property. If the number of numeric placeholders does not equal 9, 5, or 6. |

Using the DateFormatter

The `DateFormatter` class gives you a wide range of combinations for displaying date and time information. The `format()` method accepts a `Date` object, which it renders to a string based on a user-defined pattern. The `format()` method can also accept a string-formatted date, which it attempts to parse into a valid `Date` object prior to formatting.

The `DateFormatter` class has a `parseDateString()` method that accepts a date formatted as a string. The `parseDateString()` method examines sections of numbers and letters in the string to build a `Date` object. The parser is capable of interpreting long or abbreviated (three-character) month names, time, am and pm, and various representations of the date. If the `parseDateString()` method is unable to parse the string into a `Date` object, it returns null.

The following examples show some of the ways strings can be parsed:

```

"12/31/98" or "12-31-98" or "1998-12-31" or "12/31/1998"
"Friday, December 26, 2003 8:35 am"
"Jan. 23, 1989 11:32:25"

```

The DateFormatter class parses strings from left to right. Dates should appear first and must be included. Time is optional. A time signature of 0:0:0 is the Date object's default for dates that are defined without a time. Days of the week and timezone offsets are not parsed.

Pattern strings

You provide the DateFormatter class with a string of pattern letters, which it parses to determine the appropriate formatting. You must understand how to compose the string of pattern letters to control the formatting options and the format of the string that is returned.

You compose a pattern string using specific uppercase letters; for example, YYYY/MM. The DateFormatter pattern string can contain other text in addition to pattern letters, but it must end with a pattern letter. Text following the last pattern letter is truncated. It is usually best to use a format that starts and ends with a pattern letter. To form a valid pattern string, you need only one pattern letter.

Pattern letters are usually repeated. The number of repeated letters determines the presentation. For numeric values, the number of pattern letters is the minimum number of digits; shorter numbers are zero-padded to this amount. In cases where there is a corresponding mapping of a text description, if the number of pattern letters is four or more, the full form is used; otherwise, a short or abbreviated form is used if available. For example, if you specify MMMM for month, the full month name is used instead of the abbreviated month name.

For time values, a single pattern letter is interpreted in one or two digits. Two pattern letters are interpreted as two digits.

The following table describes each of the available pattern letters:

| Pattern letter | Description |
|----------------|---|
| Y | Year. If the number of pattern letters is two, the year is truncated to two digits; otherwise, it appears as four digits. The year can be zero-padded, as the third example shows in the following set of examples: Examples: YY = 03 YYYY = 2003 YYYYY = 02003 |
| M | Month in year. The format depends on the following criteria: <ul style="list-style-type: none">• If the number of pattern letters is one, the format is interpreted as numeric in one or two digits.• If the number of pattern letters is two, the format is interpreted as numeric in two digits.• If the number of pattern letters is three, the format is interpreted as short text.• If the number of pattern letters is four, the format is interpreted as full text. Examples: M = 7 MM= 07 MMM=Jul MMMM= July |

| Pattern letter | Description |
|----------------|--|
| D | Day in month. Examples: D=4 DD=04 DD=10 |
| E | Day in week. The format depends on the following criteria: <ul style="list-style-type: none"> • If the number of pattern letters is one, the format is interpreted as numeric in one or two digits. • If the number of pattern letters is two, the format is interpreted as numeric in two digits. • If the number of pattern letters is three, the format is interpreted as short text. • If the number of pattern letters is four, the format is interpreted as full text. Examples: E = 1 EE = 01 EEE = Mon EEEE = Monday |
| A | AM/PM indicator. |
| J | Hour in day (0-23). |
| H | Hour in day (1-24). |
| K | Hour in am/pm (0-11). |
| L | Hour in am/pm (1-12). |
| N | Minute in hour. Examples: N = 3 NN = 03 |
| S | Second in minute. |
| Other text | You can add other text into the pattern string to further format the string. You can use punctuation, numbers, and all lowercase letters. You should avoid uppercase letters because they may be interpreted as pattern letters. Example: EEEE, MMM. D, YYYY at H:NN A = Tuesday, Sept. 8, 2003 at 1:26 PM |

The following table shows sample pattern strings and the resulting presentation:

| Pattern | Result |
|------------------------|------------------------|
| YYYY.MM.DD at HH:NN:SS | 2003.07.04 at 12:08:56 |
| EEE, MMM D, 'YY | Wed, Jul 4, '03 |
| H:NN A | 12:08 PM |
| HH o'clock A | 12 o'clock PM |
| K:NN A | 0:08 PM |

| Pattern | Result |
|--------------------------|--------------------------|
| YYYYY.MMMM.DD. JJ:NN A | 02003.July.04. 12:08 PM |
| EEE, D MMM YYYY HH:NN:SS | Wed, 4 Jul 2003 12:08:56 |

Example

The following example shows the `DateFormatter` class in an MXML file:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">

    <!-- Declare a DateFormatter and define parameters. -->
    <mx:DateFormatter id="DateDisplay" formatString="MMMM D, YYYY" />

    <!-- Call the format() method with an empty parameter to assign today's
    date. -->
    <mx:TextInput text="Today's date is {DateDisplay.format()}." />

</mx:Application>
```

At runtime, the following text is displayed:

Today's date is September 17, 2003.

Error handling

If an error occurs, an empty string is returned and a description of the error is saved to the `error` property. An error points to a problem with the value being submitted or the format string containing the user settings, as described in the following table:

| Error type | When error occurs |
|----------------|--|
| Invalid value | A value that is not a Date object is passed to the <code>format()</code> method. (An empty argument is allowed.) |
| Invalid format | <ul style="list-style-type: none"> The <code>formatString</code> property is set to empty (""). There is less than one pattern letter in the <code>formatString</code> property. |

Using the SwitchSymbolFormatter

The `SwitchSymbolFormatter` class is a utility class meant to be used mainly when creating your own custom formatters. The `SwitchSymbolFormatter` class's `formatString` property accepts a formatted string as a definition of the format pattern. You can mix alphanumeric characters and placeholder characters in this format pattern. The pattern can contain any characters that are constant for all values of the numeric portion of the string. However, the value for formatting must be numeric.

By default, the `formatString` property of the `SwitchSymbolFormatter` class uses a number sign (#) to indicate placeholder characters. You can define a different placeholder symbol by passing it to the constructor as you instantiate the formatter object.

The number of digits supplied in the source value must match the number of digits defined in the pattern string. This is the responsibility of the script calling the `SwitchSymbolFormatter` object.

Note: You can only use the `SwitchSymbolFormatter` for numeric input values. Any nonnumeric characters in the resulting formatted strings must be constant from value to value and included in the pattern string rather than the string to be formatted.

Example

You can display a series of student IDs composed of the string “SAS” (for School of Arts and Sciences) followed by a varying four-digit number by setting the `formatString` property to “SAS####” and passing the formatter a variable containing a four-digit number.

For examples of custom formatters that use the `SwitchSymbolFormatter` for their formatting logic, see [“Creating a custom formatter” on page 738](#).

Error handling

Unlike other formatters, `SwitchSymbolFormatter` does not place its error messages into an error property. Instead, it is your responsibility to test for error conditions and return an error message if appropriate.

Creating a custom formatter

You can create a custom formatter by creating a class that extends the `mx.formatters.Formatter` class or one of the standard formatters, which all extend `mx.formatters.Formatter`. Like the standard formatters, a custom formatter contains a public `format()` method that takes a value and returns a `String`. It might also contain a `formatString` property, depending on the type of value you want to format. This property provides a pattern that your formatted string will match. For example, the `NumberFormatter` and `CurrencyFormatter` classes do not have `formatString` properties, but the `ZipCodeFormatter` class does, because it formats its value based on a set pattern.

For all formatters, when an error occurs, an empty string is returned and a description is saved to an `error` property. The `error` property is inherited from the `Formatter` superclass.

To use a custom formatter component in an application, the component must be in the `ActionScript` classpath. For more information, see [Chapter 14, “Creating ActionScript Components,” on page 359](#).

The examples in the following sections use the `SwitchSymbolFormatter`, which is a utility that takes numeric input strings and parses them through a set pattern. The pattern can contain any characters as long as they are constant for all values of the numeric portion of the string.

Creating a simple formatter that displays Social Security Numbers

The custom formatter component in the following example formats nine-digit Social Security numbers:

```
import mx.formatters.Formatter
import mx.formatters.SwitchSymbolFormatter

class CustomFormatter extends Formatter
{
    private var defaultFormatString : String = "###-##-####";
```

```

// Public Properties

// Set the inspectable metadata.
// Then declare the variable to hold the pattern string.
// The declaration must be on the line immediately after the metadata.

[Inspectable(defaultValue="###-##-####")]
public var formatString : String;

// Methods

public function format( value ):String
{
    // 1. Validate value - must be a 9-digit number.
    // You must explicitly check if the value is a number.
    // The formatter does not do that for you.

    if( isNaN( value ) || value.toString().length != 9 )
        error=defaultInvalidValueError;
    return ""

    // 2. Validate format - must contain 9 number placeholders.
    // Checks to see if the formatString has nine digits.

    var numCharCnt = 0;
    for( var i=0; i<formatString.length; i++ )
        if( formatString.charAt(i) == "#" )
            numCharCnt++;

    if( numCharCnt != 9 )
        error=defaultInvalidFormatError;
    return ""

    // 3. If the formatString and value are valid, format the number.

    formatString = getParameter( "formatString", defaultFormatString );
    var dataFormatter = new SwitchSymbolFormatter();
    return dataFormatter.formatValue( formatString, value );
}
}

```

Extending the ZipCodeFormatter for nine-digit ZIP codes

The custom formatter component in the following example extends the ZipCodeFormatter class by allowing an extra format pattern (#####*#####):

```

import mx.formatters.Formatter
import mx.formatters.ZipCodeFormatter
import mx.formatters.SwitchSymbolFormatter

class ExtendedFormatter extends ZipCodeFormatter
{
    private var extendedFormatString : String = "#####*#####";
}

```

```

// Public Properties

// Set the inspectable metadata.
// Then declare the variable to hold the pattern string.
// The declaration must be on the line immediately after the metadata.

[Inspectable(defaultValue="#####*#####")]
public var formatString : String;

// Methods

public function format( value ):String
{
    formatString = getParameter( "formatString", extendedFormatString );

    // 1. If the formatString is our new pattern, then validate and format it.

    if( formatString == extendedFormatString ){

        if( String( value ).length == 5 )
            value = String( value ).concat("0000");

        if( String( value ).length == 9 ){
            var dataFormatter = new SwitchSymbolFormatter();
            return dataFormatter.formatValue( formatString, value );
        }else{
            error=defaultInvalidValueError;
            return ""
        }
    }

    // 2. Or call super and validate and format as usual.
    // This section invalidates any pattern string that doesn't fit.
    // valid United States Zip Code formats by ignoring it.
    // It uses the included ZipCodeFormatter logic instead.

    return super.format(value);
}
}

```

Formatting patterns with the number sign

The `SwitchSymbolFormatter` uses a number sign (#) to indicate a number substitution within its pattern format by default. However, there might be some cases where you want to include a number sign in your actual pattern string. In that case, you must use a different symbol to indicate a number substitution slot within the string. You can select any character you want for this alternate symbol as long as it doesn't appear within the pattern string.

The custom formatter in the following example takes apartment numbers and adds a number sign before them:

```

import mx.formatters.Formatter
import mx.formatters.SwitchSymbolFormatter

```

```

class NumberPrependFormatter extends Formatter
{

    private var defaultFormatString : String = "&&&";

    // Public Properties

    // Set the inspectable metadata.
    // Then declare the variable to hold the pattern string.
    // The declaration must be on the line immediately after the metadata.

    [Inspectable(defaultValue="#&&&")]
    public var formatString : String;

    // Methods

    public function format( value ):String
    {
        // 1. Validate value - must be a 3-digit number.
        // You must explicitly check if the value is a number.
        // The formatter does not do that for you.

        if( isNaN( value ) || value.toString().length != 3)
            error=defaultInvalidValueError;
            return ""

        // 2. Validate format - must contain 3 number placeholders.
        // Checks to see if the formatString has three digits.
        // In this case it is extremely likely that the default pattern is used.
        // But you should always include this type of check to be safe.

        var numCharCnt = 0;
        for( var i=0; i<formatString.length; i++ )
            if( formatString.charAt(i) == "&" )
                numCharCnt++;

        if( numCharCnt != 3)
            error=defaultInvalidFormatError;
            return ""

        // 3. If the formatString and value are valid, format the number.
        // Note that the constructor is passed an ampersand (&).
        // This tells the formatter to use that symbol for a number placeholder.
        // And allows you to specify the # sign as a constant to be displayed.

        formatString = getParameter( "formatString", defaultFormatString );
        var dataFormatter = new SwitchSymbolFormatter("&");
        return dataFormatter.formatValue( formatString, value );
    }
}

```


PART IV

Advanced Application Development and Debugging

This part describes how to debug and profile your Macromedia Flex applications, add MXML code to your JSP pages, and create custom HTML wrappers for your Flex applications.

The following chapters are included:

| | |
|---|-----|
| Chapter 33: Debugging Flex Applications | 745 |
| Chapter 34: Profiling ActionScript. | 769 |
| Chapter 35: Using the Flex JSP Tag Library. | 781 |

CHAPTER 33

Debugging Flex Applications

Debugging applications can be a difficult and time-consuming task during application development. To assist you in debugging your application, Macromedia Flex includes support for debug and warning messages, an error-reporting mechanism, and a command-line ActionScript debugger.

This chapter describes how to use these tools to debug your application.

Contents

| | |
|---|-----|
| About debugging | 745 |
| Enabling debug and warning messages | 746 |
| Using the error-reporting mechanism | 747 |
| Supported errors | 750 |
| About the debugger | 752 |
| Configuring the debugger | 754 |
| Invoking the debugger | 755 |
| Using the debugger | 757 |
| Debugger example | 767 |

About debugging

One of the most important aspects of debugging is gathering the necessary diagnostic information that you need to locate the cause of a problem. Flex includes several different mechanisms that you use when debugging an application, including the following:

- A warning and debug messaging mechanism that lets you control the types of messages that Flex generates while you are debugging an application.

For more information, see [“Enabling debug and warning messages” on page 746](#).

- An error-reporting mechanism built into Macromedia Flash Debug Player that lets you direct error messages to a log file. Flash Debug Player can also capture the output of the `trace()` function and write it to the log file. When debugging an application, you can use the `trace()` function to determine if your application reaches a particular line of code, write the value of a variable to the log file, or write other status information to the log file.

For more information, see [“Using the error-reporting mechanism” on page 747](#).

- A command-line ActionScript debugger, `fdb`, which is supported only on Microsoft Windows XP and Windows 2000.

For more information, see [“About the debugger” on page 752](#).

Enabling debug and warning messages

Flex provides you with control over the output of warning and debug messages. When debugging, you can enable message output to aid you in locating and fixing problems in your application. The settings that you use to control messages are defined in the `flex_app_root/WEB-INF/flex/flex-config.xml` file.

To generate any debug information, Flex must not be running in production mode. By default, the `<production-mode>` property in the `flex-config.xml` file is set to `false`, which will generate debugging information. For more information, see [Chapter 36, “Administering Flex,” on page 795](#).

You can enable data services messages using the following tags:

- `<web-service-proxy-debug>`
- `<http-service-proxy-debug>`
- `<remote-objects-debug>`

If you enable these messages, they are written to the console window of your J2EE server. In addition, if you use Flash Debug Player and enable reporting so that Flash Debug Player writes messages to the log file, you can also capture the data services messages in the log file. For more information on using error reporting, see [“Using the error-reporting mechanism” on page 747](#).

The following table describes the debugging settings in `flex-config.xml`:

| Tag | Description |
|---|---|
| <code><create-compile-report></code> | Set to <code>true</code> to generate a compiler report. This report contains the stack tree for all dependent symbols used in the application. Flex stores it in the same directory as the MXML file as <code>app_name-report.xml</code> . The default value is <code>false</code> . |
| <code><http-service-proxy-debug></code> | Set to <code>true</code> to display the HTTP proxy request and response on the server side. The default value is <code>false</code> . |
| <code><remote-objects-debug></code> | Set to <code>true</code> to display the remote object request and response on the server side as well as debug information in client-side tracing. The default value is <code>false</code> . |

| Tag | Description |
|---|---|
| <code><show-all-warnings></code> | Set to <code>true</code> to show all compiler warnings in the browser. You can override this setting by appending <code>?showAllWarnings=true false</code> to the query string. The default value is <code>true</code> . |
| <code><show-binding-warnings></code> | When you set <code><show-all-warnings></code> to <code>true</code> , this value controls whether binding warnings are shown in the browser. When <code><show-all-warnings></code> is <code>false</code> , this value has no effect. You can override this setting by appending <code>?showBindingWarnings=true false</code> to the query string. The default value is <code>true</code> . |
| <code><show-override-warnings></code> | When you set <code><show-all-warnings></code> to <code>true</code> , this value controls whether compiler override warnings are shown in the browser. When <code><show-all-warnings></code> is <code>false</code> , this value has no effect. The default value is <code>true</code> . |
| <code><show-source-in-compiler-errors></code> | Set to <code>true</code> to display source code context lines in the error pages. The default value is <code>true</code> . |
| <code><show-stacktraces-in-browser></code> | Set to <code>true</code> to display stack traces in the browser. The default value is <code>true</code> . Note: To display AMF stack traces in the browser, you must set <code><show-stacktraces></code> to <code>true</code> in the <code>gateway-config.xml</code> file. |
| <code><web-service-proxy-debug></code> | Set to <code>true</code> to display the web service proxy request and response on the server side as well as debug information in client-side tracing. The default value is <code>false</code> . |

Using the error-reporting mechanism

The error-reporting mechanism of Flash Debug Player helps you locate many types of errors in your application, including programming errors, corrupt data errors, and network errors. When Flash Debug Player encounters an error, and you have enabled error reporting, it writes an error message to the log file.

Note: To use the Flex error-reporting mechanism, you must first install Flash Debug Player. For more information, see the Flex installation instructions.

By default, the log file is named *flashlog.txt* and is located in `C:\` in Microsoft Windows 2000, or `C:\Documents and Settings\userName` in Microsoft Windows XP. Each time you start an application, the existing log file is deleted and a new one is generated.

The following versions of Flash Debug Player support the error-reporting mechanism:

- Windows versions
- Windows ActiveX control version
- Windows Netscape/Mozilla Plugin versions

Error reporting example

Flash Debug Player can detect and log several types of errors commonly encountered in ActionScript. For example, one common ActionScript error is to reference an undefined variable, shown in the following function, where the variable `varX` is not defined:

```
function calculateDiscount(price:Number) {  
    var newPrice = price*varX;  
    ...  
}
```

In Flash Player, this declaration would silently fail. However, if you use Flash Debug Player, and enable error reporting, Flash Debug Player writes the following error message to the `flashlog.txt` log file:

Warning: Reference to undeclared variable, 'varX'

The following example shows an object, `g`, that attempts to call an undefined function, `h()`:

```
var g = new Object;  
g.h();
```

In this example, Flash Debug Player writes the following message to the log file:

Warning: h is not a function

For a list of supported ActionScript errors, see [“ActionScript errors” on page 750](#).

Using the `trace()` function

You can use Flash Debug Player to capture output from the `trace()` function and write that output to the log file. Often, you use the `trace()` function when debugging applications to write a checkpoint message to signal that your application reached a specific line of code, or to output the value of a variable. In the following example, the `trace()` function writes a message to the log file when your application enters the `initDataGrid()` function:

```
function initDataGrid(numColumns:Number) {  
    trace("Made it to initDataGrid");  
    trace("In initDataGrid, numColumns = " + numColumns);  
    ...  
}
```

To write `trace()` statements to the log file, you must conform to the following rules:

- Set `TraceOutputFileEnable` to 1 in your `mm.cfg` file.
- Use Flash Debug Player.

Note: The `trace()` statements still generate output, even if Flex is in production mode.

For more information, see [“Configuring Flash Debug Player” on page 749](#).

Error types

All versions of Flash Debug Player support ActionScript errors. These errors are typically associated with programming errors in ActionScript. For a complete list of these errors, see [“ActionScript errors” on page 750](#).

All versions of Flash Debug Player also support a collection of miscellaneous errors, such as a depth conflict. For a list of these errors, see [“Other errors” on page 751](#).

The ActiveX player and Netscape/Mozilla versions of Flash Debug Player delegate many network tasks to the browser; that is, Microsoft Internet Explorer and Netscape handle the HTTP requests, FTP requests, and socket connections for the Player. The browsers do not report back to the Player or inform the user of any HTTP status codes, network errors, and so on. These error types will not be reported by the ActiveX player and Netscape/Mozilla versions of Flash Debug Player.

The stand-alone Flash Debug Player implements the HTTP, FTP, and socket functionality. Therefore, these errors will be reported in the stand-alone Flash Debug Player. For more information on these errors, see the following sections:

- [“HTTP errors” on page 751](#)
- [“FTP errors” on page 752](#)
- [“Network errors” on page 752](#)

Configuring Flash Debug Player

You configure the error-reporting mechanism of Flash Debug Player using the `mm.cfg` text file. This file is typically located in the same directory as the `flashlog.txt` file.

Two Microsoft Windows environment variables define the location of this directory:

HOMEDRIVE Specifies the drive letter of the path to the home directory. In most Microsoft Windows systems, the default value is `C:`, the primary hard drive.

HOMEPATH Specifies the path to the home directory, relative to **HOMEDRIVE**. On Microsoft Windows 2000, the default is `\`. In Microsoft Windows XP, the default is `\Documents and Settings\user_name` where *user_name* is your system user name.

Therefore, the default home directory is `C:\` in Microsoft Windows 2000 and `C:\Documents and Settings\user_name` in Microsoft Windows XP.

For information on setting these two environment variables, see the Flex installation instructions.

The following `mm.cfg` file enables error reporting and configures Flash Debug Player to write error messages to the `flashlog.txt`, where `flashlog.txt` is in the same directory as `mm.cfg`:

```
ErrorReportingEnable=1
TraceOutputFileEnable=1
```

The following table lists the properties that you can set in the mm.cfg file:

| Property | Description |
|-----------------------|--|
| ErrorReportingEnable | Set <code>ErrorReportingEnable</code> to 1 to enable error reporting. Disable it by setting it to 0. Error reporting is disabled by default. |
| MaxWarnings | Set <code>MaxWarnings</code> to override the default message limit. For example, you can set it to 500 to capture 500 error messages. By default, Flash Debug Player logs 100 error messages to the log file. After 100 messages, Flash Debug Player writes a message to the file stating that further error messages will be suppressed. Set the <code>MaxWarnings</code> to 0 to remove the limit so that all error messages are recorded. |
| TraceOutputFileEnable | Set <code>TraceOutputFileEnable</code> to 1 to enable Flash Player to write error messages to the log file. Disable it by setting <code>TraceOutputFileEnable</code> to 0. |
| TraceOutputFileName | Set <code>TraceOutputFileName</code> to override the default name and location of the log file by specifying a new location and name in the form: <code>TraceOutputFileName=<fully qualified path/filename></code> By default, Flash Player writes error messages to a file named flashlog.txt located in the directory specified by the environment variables HOMEDRIVE and HOMEPATH. |

Supported errors

The following sections list the error types supported by Flash Debug Player. Some error types are supported by all versions of Flash Debug Player, while others are supported by the Windows Standalone Flash Debug Player.

ActionScript errors

The following ActionScript errors are supported by all versions of the Flash Debug Player:

| Symbolic name | Message string | Code snippet | Note |
|--------------------|---|---|---|
| NOT_DEFINED | {0} is not defined where {0} is the name of the object causing the error | <code>varX;</code> | Assigning a value to <code>varX</code> implicitly defines it so that is not an error. Also, accessing a member that is not defined for an object is not an error. |
| NO_PROPERTIES | {0} has no properties | <code>var varX; varX.varY;</code> | The following is not an error: <code>var varX = new Object; varX.varY;</code> |
| NOT_FUNCTION | {0} is not a function | <code>var varX = new Object; varX.funcY();</code> | |
| UNCAUGHT_EXCEPTION | uncaught exception: {0} | <code>throw new Error;</code> | |
| STACK_OVERFLOW | stack overflow | | |

Other errors

The following table lists errors supported by all versions of Flash Debug Player:

| Symbolic name | Message string |
|-------------------|--------------------------------------|
| DepthConflict | Failed to place object at depth. |
| CorruptData | Failed to parse corrupt data. |
| DownloadIsUnknown | Not a known player download type: %s |

HTTP errors

The following HTTP errors are supported by the Windows Flash Debug Player. For more information on these errors, see www.w3.org/Protocols/rfc2616/rfc2616-sec10.html.

| Symbolic name | Message string | Build/configuration |
|---------------|--|---------------------------------------|
| 400 | Bad Request | Windows Standalone Flash Debug Player |
| 401 | Unauthorized | Windows Standalone Flash Debug Player |
| 402 | Payment required | Windows Standalone Flash Debug Player |
| 403 | Forbidden | Windows Standalone Flash Debug Player |
| 404 | Not found | Windows Standalone Flash Debug Player |
| 405 | Method Not Allowed, Invalid MIME type | Windows Standalone Flash Debug Player |
| 406 | Not Acceptable | Windows Standalone Flash Debug Player |
| 500 | Server Error | Windows Standalone Flash Debug Player |
| 501 | Not Implemented | Windows Standalone Flash Debug Player |
| 502 | Bad Gateway | Windows Standalone Flash Debug Player |
| 503 | Out of Resources | Windows Standalone Flash Debug Player |
| 504 | Gateway Time-Out | Windows Standalone Flash Debug Player |
| 505 | HTTP Version not supported | Windows Standalone Flash Debug Player |
| | Internet Open error | Windows Standalone Flash Debug Player |
| | Internet Connect error: %s | Windows Standalone Flash Debug Player |
| | No URL Path | Windows Standalone Flash Debug Player |
| | HTTP Open Request error: %s | Windows Standalone Flash Debug Player |
| | HTTP Send Request error %d: %s | Windows Standalone Flash Debug Player |
| | HTTP Query Info error: %s | Windows Standalone Flash Debug Player |

FTP errors

The following FTP errors are supported by the Windows Standalone Debug Player:

| Message string | Build/configuration |
|-------------------------|---------------------------------------|
| FTP open File error: %s | Windows Standalone Flash Debug Player |

Network errors

The following network errors are supported by the Windows Standalone Flash Debug Player:

| Symbolic name | Message string | Build/configuration |
|---------------|----------------------------|---------------------------------------|
| ConnectError | Cannot resolve address: %s | Windows Standalone Flash Debug Player |
| ConnectError | Could not connect: %s:%d | Windows Standalone Flash Debug Player |
| SocketError | Unable to resolve host: %s | Windows Standalone Flash Debug Player |
| SocketError | Socket Error: %s:%d | Windows Standalone Flash Debug Player |
| SocketError | Could not connect: %s:%d | Windows Standalone Flash Debug Player |

About the debugger

The Flex ActionScript command-line debugger, `fdb`, lets you step through and debug ActionScript files used by your Flex applications. Some of these ActionScript files are external class files, and some are generated by the Flex server when it compiles the application.

There can be any number of files per application, but in general Flex generates a single file containing ActionScript statements used in `<mx:Script>` blocks within the MXML file, and an additional file for each ActionScript class file used by that application.

You can use `fdb` in any gdb-compatible debugging environment. For example, you can use M-x `gdb` inside Emacs and specify `fdb.exe` as the `gdb` command.

For an overview of the `fdb` debugger, use the following tutorial command:

```
(fdb) tutorial
```

Working with ActionScript files

ActionScript is the foundation of Flex components. During compilation, Flex links code from system classes (contained within SWC files), your code (contained within AS files), third-party SWC files, and generated code (the result of compiling your MXML files into ActionScript statements). The debugger uses these files to step through the application.

Two generated files are produced for each compiled MXML file; these files are not visible on disk and can only be viewed within the debugger.

The first file, *compiled_mxml_file_name.1* (where *compiled_mxml_file_name* is the name of the MXML file that was compiled), contains the class definition of the class associated with the compiled MXML. In addition, it contains an `init()` function, event handler, and any other code required to implement the MXML. You can set breakpoints on most functions in this file; the exception is event handlers. You can set breakpoints for the event handlers only in the original MXML file.

The second file, *compiled_mxml_file_name.2*, contains the deferred instantiation entry for the MXML mapped class, if any.

Note: These files are not the same as the *compiled_mxml_file_name-generated.as* file that is optionally written to disk during compilation.

In `fdb`, you can list the files using the `info` command. For more information, see [“Getting status” on page 765](#).

The main MXML file for the application appears in the list first, with the others following in Unicode order (mixing ActionScript and MXML files). File numbers begin at 1 and are sequential. The application includes the following files:

- ActionScript files for Frameworks and system class files
- ActionScript files for generated files
- ActionScript files for authored files such as external custom classes

About SWD files

To debug a Flex application, you must generate a SWD (Flash Debug) file. SWD files are similar to SWF files, except that they contain debugging-specific information that the debugger and Flash Debug Player watch for.

When Flex generates a SWD file from the MXML file, it stores the SWD file in the same directory as the SWF file (the MXML file’s current directory). These files must remain in the same directory for debugging. If the SWD file does not exist or does not match the SWF file, the debugger returns an error.

The debugger generates a SWD file for you when it invokes the default browser. To debug in the stand-alone Flash Player, you must first pregenerate the SWD file using either the `mxmmlc` compiler or by requesting the MXML file in a browser. If you request the file in a browser, you must either append `?debug=true` to the query string or set the `<generate-debug-swfs>` option to `true` in the `flex-config.xml` file.

To debug a custom component, you can add a SWD file to the component’s SWC file when you export it from the Flash MX 2004 development environment.

Debugger limitations

The debugger supports only ActionScript-level debugging and does not support the Flash Timeline concept. The debugger also does not allow debugging MXML tags, with the exception of setting breakpoints on event handlers defined on MXML tags.

In a Flex environment, Macromedia Flash Player may interact with the server. The debugger does not assist in debugging the server-side portion of the application, nor does it offer support for inspecting any of the IP transactions that take place from Flash Player to the server, and vice versa.

Debugger shortcuts

You can invoke commands within the fdb debugger using the fewest number of nonambiguous keystrokes. For example, to use the print command, you can type **p**, because no other command begins with that letter.

Configuring the debugger

You can run the ActionScript debugger with the debug version of Macromedia Flash Player registered with your default browser (Windows only) or in the stand-alone Flash Player (Windows and UNIX).

The fdb debugger is installed in the *flex_install_dir/bin* directory. To start fdb, navigate to this directory and type **fdb** at the command line.

For more information on installing Flash Debug Player or installing the stand-alone Flash Player, see the installation documentation.

Changing global debugger settings

You configure the debugger using child tags of the `<debugging>` tag in the `flex-config.xml` file. Settings in the configuration file affect all debugging sessions. You can override settings in the `flex-config.xml` file with a query string parameter unless `<production-mode>` is `true`.

The following table describes the fdb-related debugging tags in the `flex-config.xml` file:

| Property | Description |
|---|---|
| <code>production-mode</code> | Set to <code>true</code> to disable all debugging options, regardless of individual settings. They cannot be overridden with query string parameters. Set to <code>false</code> to allow debugging options. |
| <code>process-debug-query-params</code> | Allows override of values in debugging section using query parameter strings on a per-request basis. |
| <code>generate-debug-swfs</code> | Generate SWF and SWD files for debugging. You can override this setting using <code>?debug=true</code> or <code>?debug=false</code> on the request string. If you invoke the debugger in the default browser, the debugger generates the SWF and SWD files for you. |
| <code>keep-generated-as</code> | Writes the generated ActionScript files to the disk. |
| <code>keep-generated-swfs</code> | Writes the generated SWF and SWD files to the disk. |

You cannot use the debugger when Flex is running in production mode. In the `flex-config.xml` configuration file, set the value of the top-level `<production-mode>` tag to `false`; for example:

```
<production-mode>false</production-mode>
```

For more information on editing the `flex-config.xml` file, see [“Editing the flex-config.xml file” on page 806](#).

Using the debugger in Windows

In Windows, you can run your Flex applications in a browser or in the stand-alone Flash Player.

When debugging an application in a web browser, fdb uses only the *default browser*. The default browser is the browser that opens when you open a web-specific file without specifying an application. You must also have Flash Debug Player installed with this browser. If you do not have the correct version of Flash Debug Player, Flash displays an error indicating that your Flash Player does not support all fdb commands.

Your default browser might not be the first browser that you installed on your computer. For example, if you installed another web browser *after* installing Internet Explorer, Internet Explorer might not be your default browser.

To determine what browser is your default browser:

1. From the Windows ToolBar, select Start.
2. Select Run.
3. Enter a URL in the Run dialog box; for example:
`http://www.macromedia.com`
4. Click OK.

Windows opens the default browser or displays an error message indicating that there is no application configured to handle your request.

To set Internet Explorer as your default browser:

1. Open the Internet Explorer application.
2. Select Tools > Internet Options.
3. Select the Programs tab.
4. Click the Reset Web Settings button.

To set Netscape 7.x as your default browser:

1. Open the Netscape application.
2. Select Edit > Preferences.
3. Click the Set Default Browser button.

Invoking the debugger

This section describes how to start a debugging session with fdb on Windows and UNIX systems. For information on what commands are available after you start a debugging session, see [“Using the debugger” on page 757](#).

For more information on editing the flex-config.xml file, see [“Editing the flex-config.xml file” on page 806](#).

Starting a session with the default browser (Windows only)

The easiest way to start a debugging session is to use `fdb` to invoke the SWD file in the default browser. To start a debug session in the default browser, you use the `fdb` command from the command line.

To start the debugger:

1. Open a console window.
2. Find the *Flex_install/bin* directory. You installed the Flex application files to this directory. It is not the same directory as the Flex deployment directory.

The default *Flex_install* directory on Windows is `C:/Program Files/Macromedia/Flex/`.

3. Type `fdb` from the command line, followed by the path to the MXML file; for example:

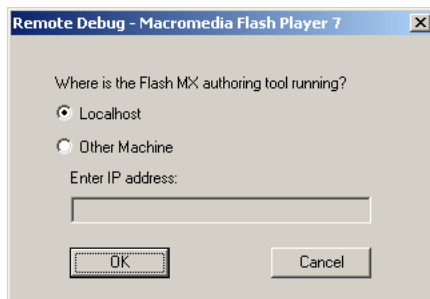
```
fdb http://localhost:8100/flex/MyApp.xml
```

The `fdb` debugger starts the Flex application in your default browser and appends `?debug=true` to the query string. This generates a SWD file if one is not already present. The (`fdb`) command prompt appears in the console window, as the following example shows:

```
Attempting to launch and connect to Player using URI
http://localhost:8100/flex/wrapper/file1.xml
Player connected; session starting.
Set breakpoints and then type 'continue' to resume the session.
(fdb)
```

If `fdb` does not connect to the player, you might not be running the Flash Debug Player. For information on installing the Flash Debug Player, see the installation instructions.

You can also start a debugging session by running a SWF file or MXML file that has an accompanying SWD file in Flash Debug Player. Or, you can start a debugging session by requesting a SWF file or MXML file in Flash Debug Player and appending `?debug=true` to the query string. In these cases, you are prompted to supply a location for the debugger utility, as the following figure shows:



If you are running `fdb` and the Flex server on the same computer, select `Localhost`. If you are running the debugger remotely, select `Other Machine` and enter the other computer's IP address (and server port number) in the Enter IP Address field.

Debugging with the stand-alone Flash Debug Player (Windows and UNIX)

Stand-alone Flash Debug Player runs as an independent application. It does not run within a web browser or other shell. Stand-alone Flash Debug Player does not support any server requests such as web services and dynamic SWF loading, so not all applications can be properly debugged inside the stand-alone Flash Debug Player.

To start a debug session with stand-alone Flash Debug Player, you must generate a SWD file (Flash Debug file) and SWF file for the application.

To debug with the stand-alone Flash Debug Player:

1. Generate the Flex application's SWD and SWF files in one of the following ways:
 - Request the MXML file in a browser and append `?debug=true` to the query string.
 - Use the following `mxmmlc` command-line compiler with the `-g` option:

```
mxmmlc -g MyApp.mxml
```

For more information on using `mxmmlc`, see [“Using the command-line compiler” on page 796](#).

2. Find the `Flex_install_dir/bin` directory. You installed the Flex application files to this directory. It is not the same directory as the Flex deployment directory.

The default `Flex_install_dir` directory in Windows is `C:\Program Files\Macromedia\Flex\`.

3. Type `fdb` from the command line, followed by the path to the SWF file; for example:

```
fdb c:/jrun4/servers/flex_server/flex/MyApp.swf
```

The `fdb` debugger starts the Flex application in the stand-alone Flash Debug Player, and the (`fdb`) command prompt appears.

Using the debugger

This section describes commands that you use to debug and navigate your Flex application using the `fdb` debugger.

Running the debugger

The `fdb` debugger provides several commands for stepping through the debugged file. The following table summarizes these commands:

| Command | Description |
|--------------------------|---|
| <code>continue</code> | Continues running the application. |
| <code>file [file]</code> | Specifies an application to be debugged, without starting it. This command does not cause the application to start; use the <code>run</code> command with no argument to start debugging the application. |
| <code>finish</code> | Continues until the function exits. |
| <code>next [N]</code> | Continues to the next source line in the application. Optional argument <i>N</i> , means do this <i>N</i> times or until the program stops for some other reason. |
| <code>quit</code> | Exits from the debug session. |

| Command | Description |
|-------------------------|---|
| <code>run [file]</code> | Starts a debugging session by running the specified file. Execute the <code>run</code> command without any options to run the application previously specified by the file command. The <code>run</code> command starts the application in a browser or stand-alone Flash Player. |
| <code>step [N]</code> | Steps into the application. Optional argument <i>N</i> , means do this <i>N</i> times or until the program stops for some other reason. |

When you start a session, the debugger stops execution before Flex renders the application on the screen. Use the `continue` command to get to the application's starting screen.

The following example shows a sample application after it starts:

```
(fdb) continue
[trace] ComboBase: y = undefined text_mc.bl = undefined
[trace] ComboBase: y = undefined text_mc.bl = undefined
[trace] ComboBase: y = undefined text_mc.bl = undefined
[trace] ComboBase: y = undefined text_mc.bl = undefined
[trace] ComboBase: y = undefined text_mc.bl = undefined
[trace] ComboBase: y = undefined text_mc.bl = undefined
[trace] ComboBase: y = undefined text_mc.bl = undefined
[trace] RadioButtonGroup.addInstance: instance =
_level10._VBox0._Accordion0._For
m2._FormItem3._RadioButton1 data = undefined label = 2003
[trace] RadioButtonGroup.addInstance: instance =
_level10._VBox0._Accordion0._For
m2._FormItem3._RadioButton2 data = undefined label = 2004
[trace] RadioButtonGroup.addInstance: instance =
_level10._VBox0._Accordion0._For
m2._FormItem3._RadioButton3 data = undefined label = 2005
[trace] RadioButtonGroup.addInstance: instance =
_level10._VBox0._Accordion0._For
m2._FormItem3._RadioButton4 data = undefined label = 2006
[trace] ComboBase: y = 0 text_mc.bl = 12
[trace] ComboBase: y = 0 text_mc.bl = 12
[trace] ComboBase: y = 0 text_mc.bl = 12
[trace] ComboBase: y = 0 text_mc.bl = 14
```

You interact with the application in the Flash Debug Player. For example, if you select an item from the drop-down list, the debugger continues to output information to the command window:

```
[trace] SSL : ConfigureScrolling
[trace] SSP : 5 51 true 47
[trace] ComboBase: y = 0 text_mc.bl = 14
[trace] layoutChildren : bRowHeightChanged
[trace] >>SSL:layoutChildren
[trace] deltaRows 5
[trace] rowCount 5
[trace] <<SSL:layoutChildren
[trace] >>SSL:draw
[trace] bScrollChanged
[trace] SSL : ConfigureScrolling
[trace] SSP : 5 51 false 46
```

[trace] SSL Drawing Rows in UpdateControl 5
[trace] <<SSL:draw

You can store commonly used commands in a source file and then load that file using the `source` command. For more information, see [“Using the source command” on page 761](#).

Using breakpoints

Setting breakpoints is a critical aspect of debugging any application. You can set breakpoints on any ActionScript code in your Flex application. You can set breakpoints on statements in any external ActionScript file, on ActionScript statements in an `<mx:Script>` tag, or on MXML tags that have event handler properties. In the following MXML code, `click` is an event handler property:

```
<mx:Button click="ws.getWeather.send();" />
```

You cannot use breakpoints in event handlers of the web service results.

Breakpoints are maintained from session to session. However, when you change the target file or quit `fdb`, breakpoints are lost.

The following table summarizes the commands for manipulating breakpoints with the ActionScript debugger:

| Command | Description |
|--|--|
| <code>break [args]</code> | Sets breakpoint at the specified line or function. The argument can be a line number or function name. With no arguments, the <code>break</code> command sets a breakpoint at the currently stopped line (not the currently listed line). If you specify a line number, <code>fdb</code> breaks at the start of code for that line. If you specify a function name, <code>fdb</code> breaks at the start of code for that function. |
| <code>clear [args]</code> | Clears breakpoint at specified line or function. The argument can be a line number or function name. If you specify a line number, <code>fdb</code> clears a breakpoint in that line. If you specify a function name, <code>fdb</code> clears a breakpoint at the beginning of that function. With no argument, <code>fdb</code> clears a breakpoint in the line that the selected frame is executing in. See the <code>delete</code> command, which clears breakpoints by number. |
| <code>commands [breakpoint]</code> | Sets commands to execute when the specified breakpoint is encountered. If you do not specify a breakpoint, the commands are applied to the first breakpoint. |
| <code>condition bnum [expression]</code> | Specifies a condition that must be met to stop at the given breakpoint. The <code>fdb</code> debugger evaluates <i>expression</i> when the <i>bnum</i> breakpoint is reached. If the value is <code>true</code> or nonzero, <code>fdb</code> stops at the breakpoint. Otherwise, <code>fdb</code> ignores the breakpoint and continues execution. To remove the condition from the breakpoint, do not specify an <i>expression</i> . You can use conditional breakpoints to stop on all events of a particular type. For example, to stop on every initialize event, use the following commands: <code>(fdb) break UIEvent:dispatch</code> <code>Breakpoint 18 at 0x16cb3: file UIEventDispatcher.as, line 190</code> <code>(fdb) condition 18 (eventObj.type == 'initialize')</code> |

| Command | Description |
|---|---|
| <code>delete [args]</code> | Deletes breakpoints. Specify one or more comma- or space-separated breakpoint numbers to delete those breakpoints. To delete all breakpoints, give no argument. |
| <code>disable breakpoints [bp_num]</code> | Disables breakpoints. Specify one or more space-separated numbers as options to disable only those breakpoints. |
| <code>enable breakpoints [bp_num]</code> | Enables breakpoints that were previously disabled. Specify one or more space-separated numbers as options to enable only those breakpoints. |

The following example sets a breakpoint on the `myFunc()` method, which is triggered when the user clicks a button:

```
(fdb) break myFunc
Breakpoint 1 at 0x401ef: file file1.mxml, line 5
(fdb) continue
Breakpoint 1, myFunc() at file1.mxml:5
   5           tal.text = "Clicked";
(fdb)
```

To see all breakpoints and their numbers, use the `info breakpoints` command.

You can use the `commands` command to periodically print out values of objects and variables whenever `fdb` encounters a particular breakpoint. The following example prints out the value of `tal.text` (referred to as `$1`), executes the `where` command, and then continues when it encounters the button's click handler breakpoint:

```
(fdb) commands 1
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just 'end'.
>print tal.text
>where
>continue
>end
(fdb) cont
Breakpoint 1, myFunc() at file1.mxml:5
   5           tal.text = "Clicked";
$1 = ""
#0  [MovieClip 1].myFunc(event=undefined) at file1.mxml:5
#1  [MovieClip 1].handler(event=[Object 18127]) at file1.mxml:15
```

Breakpoints are not specific to a single SWF file. If you set a breakpoint in a file that is common to multiple SWF files, `fdb` applies the breakpoint to all SWF files.

For example, suppose you have four SWF files loaded and each of those SWF files contains the same version of an ActionScript file, `view.as`. If you want to set a breakpoint in the `init()` function of the `view.as` file, you are only required to set a single breakpoint in one of the `view.as` files. When `fdb` encounters any of the `init()` functions, it triggers the break.

Using watchpoints

Watchpoints identify variables that trigger the fdb debugger to stop when those variables are accessed. You can set watchpoints to trigger when the variables are read, written to, or both read and written to.

The `watch` command has the following syntax:

```
(fdb) watch [expression]
```

The *expression* is any variable or property. The following example stops fdb when the `color` property of the `myButton` object changes:

```
(fdb) watch myButton.color
```

Use the `awatch` command to stop fdb when the variable is written to or read from. Use the `rwatch` command to stop fdb when the variable is read.

Watchpoints are a variant of breakpoints and can be manipulated with a subset of breakpoints commands. Use `info breakpoints` to list all watchpoints (in addition to breakpoints). To delete all watchpoints, use the `delete` command.

Using the source command

You can use the `source` command to read fdb commands from a file and execute them. This lets you write commands such as breakpoints once and use them repeatedly when debugging the same application in different sessions or across different applications.

The `source` command has the following syntax:

```
(fdb) source file
```

The value of *file* can be a filename for a file in the current working directory or an absolute path to a remote file. To determine the current working directory, use the `pwd` command.

The following examples read in the `mycommands.txt` file from different locations:

```
(fdb) source mycommands.txt
(fdb) source mydir\mycommands.txt
(fdb) source c:\mydir\mycommands.txt
```

Examining data values

The `print` command displays values of variables, objects, and properties. To view the context of the variable, use the `what` command. You can save a list of common variables that you want fdb to display using the `display` command.

The `print` command uses the following syntax:

```
print [variable_name | object_name[.] | property]
```

The `print` command prints the value of the specified variable, object, or property. You can specify the `name` or `name.property` to narrow the results. If fdb can determine the type of the entity, fdb displays the type.

If `print` is performed on an object, fdb displays a numeric identifier for the object.

Note: The `print` command does not support expressions.

To list all the properties of an object, use trailing dot-notation syntax. The following example prints all the properties of the object `myButton`:

```
(fdb) print myButton.
```

To print the value of a single variable, use dot-notation syntax, as the following example shows:

```
(fdb) print myButton.color
```

Use the `what` command to view the context of a variable. The `what` command has the following syntax:

```
(fdb) what variable
```

Use the `display` command to add an expression to the auto-display list. Every time debugging stops, `fdb` prints the list of expressions in the auto-display list. The `display` command has the following syntax:

```
(fdb) display [expression]
```

The expression is the same as the arguments for the `print` command; for example:

```
(fdb) display myButton.color
```

To view all expressions on the auto-display list, use the `info display` command.

To remove an expression from the auto-display list, use the `undisplay` command. The `undisplay` command has the following syntax:

```
(fdb) undisplay [list_num]
```

Use the `undisplay` command without an argument to remove all entries on the auto-display list. Specify one or more `list_num` options separated by spaces to remove numbered entries from the auto-display list.

You can temporarily disable auto-display expressions using the `disable display` command. The `disable` command has the following syntax:

```
(fdb) disable display [display_num]
```

Specify one or more space-separated numbers as options to disable only those entries in the auto-display list.

To re-enable the display list, use the `enable display` command, which has the same syntax as the `disable display` command.

Changing data values

You can use the `set` command to set the value of a variable or a convenience variable. The `set` command has the following syntax:

```
set [expression]
```

Depending on the variable type, you use different syntax for the *expression*. The following example sets the variable `i` to the number 3:

```
(fdb) set i = 3
```

The following example sets the variable `employee.name` to the string `Susan`:

```
(fdb) set employee.name = "Susan"
```

The following example sets the convenience variable `$myVar` to the number 20:

```
(fdb) set $myVar = 20
```

Convenience variables are variables that exist entirely within fdb; they are not part of your application. Convenience variables are prefixed with `$` and can have any name that does not conflict with any existing variable.

The following table describes pre-existing convenience variables used by fdb:

| Variable | Description |
|----------------------------------|--|
| <code>\$listsize</code> | Sets the number of lines to display with the <code>list</code> command. The default value is 10. |
| <code>\$columnwrap</code> | Sets the column number on which output wraps in the display. The default is value <code>undefined</code> . |
| <code>\$infostackshowthis</code> | Set to 0 to suppress <code>this</code> in stack backtraces. The default value is 1. |
| <code>\$invokegetters</code> | Set to 0 to prevent fdb from firing getter functions. The default value is 1 (enabled). |
| <code>\$bpnum</code> | Displays the last defined breakpoint number. |

Viewing file contents

Use the `list` command to view lines of code in the `ActionScript` files. The `list` command uses the following syntax:

```
list [- | line_num[,line_num] | [file_name:]line_num | file_name[:line_num] |  
      [file_name:]function_name]
```

The `list` command prints the lines around the specified function or line of the current file. If you do not specify an argument, `list` prints 10 lines after or around the previous listing. If you specify a filename, but not a line number, `list` assumes line 1.

To set the list location to where the execution is currently stopped, use the `home` command.

If you specify a single numeric argument, the `list` command lists 10 lines around that line. If you specify more than one comma-separated numeric argument, the `list` command displays lines between and including those line numbers.

The following example lists code from line 10 to line 15:

```
(fdb) list 10, 15
```

If you specify a hyphen (-) in the previous example, the `list` command displays the 10 lines before a previous 10-line listing.

Specify a line number to list around that line in the current file; for example:

```
(fdb) list 10
```

Specify a filename followed by a line number to list around that line in that file; for example:

```
(fdb) list effects.xml:10
```

Specify a function name to list the lines around the beginning of that function; for example:

```
(fdb) list myFunction
```

Specify a filename followed by a function name to list the lines around the beginning of that function. This lets you distinguish among like-named static functions; for example:

```
(fdb) list effects.mxml:myFunction
```

You can resolve ambiguous matches by extending the value of the function name or filename, as the following examples show:

Filenames:

```
(fdb) list UIOb
Ambiguous matching file names:
UIObject.as#66
UIObjectDescriptor.as#67
UIObjectExtensions.as#68
(fdb) list UIObject.
```

Function names:

```
(fdb) list init
Ambiguous matching function names:
init
initFromClipParameters
(fdb) list init(
```

Viewing and changing the current file

The `list` command acts on the current file by default. To change to a different file, use the `cf` command. The `cf` command has the following syntax:

```
(fdb) cf [file_name|file_number]
```

For example, to change the file to `MyApp.mxml`, use the following command:

```
(fdb) cf MyApp.mxml
```

If you do not specify a filename, the `cf` command lists the name and file number of the current file.

You can also use the `viewswfs` command to all files associated with the current SWF file. For more information, see the online help.

Viewing the current working directory

Use the `pwd` command to view the file system's current working directory. This is the directory from which `fdb` was run; for example:

```
(fdb) pwd
c:/Program Files/Macromedia/Flex/bin/
```

Using truncated file and function names

The `fdb` debugger supports truncated file and function names. You can specify `file_name` and `function_name` arguments with partial names, as long as the names are unambiguous.

If you use truncated file and function names, fdb tries to map the argument to an unambiguous function name first, then a filename. For example, `list foo` first tries to find a function unambiguously starting with "foo" in the current file. If this fails, it tries to find a file unambiguously starting with "foo".

Printing stack traces

Use the `bt` command to display a back trace of all stack frames. The `bt` command has the following syntax:

```
(fdb) bt
```

Getting status

Use the `info` command to get general information about the application. The `info` command has the following syntax:

```
info [arguments|breakpoints|files|functions|locals|sources|swfs|targets|
    variables] [args]
```

The `info` command displays generic information about the program being debugged. The following table describes the options of the `info` command:

| Option | Description |
|-----------------|--|
| arguments | Displays the argument variables of the current stack frame. |
| breakpoints | Displays the status of user-settable breakpoints. |
| display | Displays the list of autodisplay expressions. |
| files [arg] | <p>Displays the names of all files used by the target application. This includes authored files and system files, plus generated files. Also indicates the file number for each file.</p> <p>You can use wildcards and literals to select and sort the output. The <code>info files</code> command supports the following:</p> <p><code>info files character</code> Alphabetically lists files with names that start with the specified <i>character</i>. The following example lists all files starting with the letter V:</p> <pre>info files V</pre> <p><code>info files *.extension</code> Alphabetically lists all files with the given extension. The following example lists all files with the <code>as</code> extension:</p> <pre>info files *.as</pre> <p><code>info files *string*</code> Alphabetically lists all files with names that include <i>string</i>.</p> |
| functions [arg] | <p>Displays all function names used in this application. The <code>info functions</code> command optionally takes an argument; for example:</p> <pre>info functions</pre> Lists all functions in all files. <pre>info functions</pre> Lists all functions in the current file. <pre>info functions MyApp.mxml</pre> Lists all functions in the <code>MyApp.mxml</code> file. |
| handle | Displays settings for fault handling in the debugger. |
| locals | Displays the local variables of the current stack frame. |
| sources | Displays authored source files used by the target application. |
| stack | Displays the backtrace of the stack. |

| Option | Description |
|-----------|--|
| swfs | Displays all current SWF files. |
| targets | Displays the HTTP or file URL of the target application. |
| variables | Displays all global and static variable names. |

Handling faults

Use the `handle` command to specify how `fdb` should react to Flash Player errors during execution. You can specify that `fdb` reacts differently depending on the type of fault.

The `handle` command has the following syntax:

```
(fdb) handle [fault_type|all] [action]
```

The `fault_type` is the category of fault that `fdb` handles. The `action` is what `fdb` does in response to that fault. To assign a single action to all fault types, use `all` for the `fault_type`. The following table describes the fault types:

| Fault type | Description |
|-----------------|--|
| exception | The application threw a user exception. |
| invalid_target | The application's <code>ActionSetTarget</code> instruction had a bad target name. |
| invalid_url | The application failed to open a URL. |
| invalid_with | The target of a <code>with</code> statement in the application is not an object. |
| proto_limit | The application's search up a prototype chain reached the limit. |
| recursion_limit | The application reached the upper bound of the recursion limit. |
| script_timeout | The ActionScript inside the application stopped running and the upper limit of wait time was exceeded. |
| stack_underflow | A stack underflow occurred. |
| zero_divide | The application encountered a divide-by-zero error. |

The possible actions are `print`, `noprint`, `stop`, and `nostop`. The following table describes these actions:

| Action | Description |
|---------|---|
| print | Prints a message if this type of fault occurs. |
| noprint | Does not print a message if this type of fault occurs. |
| stop | Stops execution of the debugger if this type of fault occurs. |
| nostop | Does not stop execution of the debugger if this type of fault occurs. |

To view the current settings, use the `info` command, as the following example shows:

```
(fdb) info handle
```

Getting help

Use the `help` command to get information on particular topics. The `help` command has the following syntax:

```
help [topic]
```

The `help` command provides a relatively terse description of each command and its usage. The following example invokes the `help` command:

```
(fdb) help
```

Type **help** followed by the command name to get the full help information, as the following example shows:

```
(fdb) help delete
```

Terminating the session

Use the `kill` and `exit` commands to end the current debugging session and exit from the `fdb` application. The `kill` and `exit` commands take no arguments. If `fdb` invoked the default browser, you can also terminate the `fdb` session by closing the browser window.

To stop the current session, use the `kill` command; for example:

```
(fdb) kill
```

Using the `kill` command does not quit the `fdb` application. You can immediately start another session. To exit from `fdb`, use the `exit` command; for example:

```
(fdb) exit
```

Debugger example

The example in this section shows the following commands:

- Using the `print` command to traverse the proto chain and identify the MXML id of the object and to probe static functions such as `_global.mx.events`.
- Using the `what` command to see how a variable reference was resolved.
- Using the `display` command to display common information every time you encounter a breakpoint.
- Creating a trace-like effect using breakpoints and the `commands` command.

```
(fdb) w
#0 [Movieclip 5279].addEventListener(event="click", handler=[Object 3058])
   at UIEventDispatcher.as:176
#1 [Movieclip 1].doFoo() at buttonClickHandler.mxml:21
#2 [Movieclip 1].handler(event=[Object 6187]) at buttonClickHandler.mxml:28
(fdb) print this
$14 = [Movieclip 5279]
(fdb) print id
$11 = "myBtn2"
(fdb) what id
this.__proto__.__proto__.__proto__.__proto__.id
(fdb) print className
$12 = "Button"
```

```

(fdb) what className
this.__proto__.className
(fdb) print label
$13 = "click me"
(fdb) print mx.events.
$16 = events = [Object 662]
EventDispatcher = [Function 664, name='_dependsOnEventDispatcher_']
EventProxy = [Function 1757]
LowLevelEvents = [Function 715, name='LowLevelEventDependency']
UIEventDispatcher = [Function 682, name='UIEventDispatcherDependency']
__constructor__ = [Function 10, name='Object']
__proto__ = [Object 11]
(fdb) what mx.events
_global.mx
(fdb) display this
(fdb) display className
(fdb) continue
Breakpoint 2, __addEventListener() at UIEventDispatcher.as:176
176 __origAddEventListener(event, handler);
4: this = [Movieclip 1]
5: this.className = "buttonClickHandler"

```

The following example sets commands on a breakpoint to create a trace-like ability. You must be sure to set additional breakpoints (possibly conditional ones) so that the application stops at some point.

```

(fdb) b UIE:dis
Breakpoint 1 at 0x401ef: file UIEventDispatcher.as, line 118
(fdb) commands
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just 'end'.
silent
p eventObj.type
p this
p this.className
c
end
(fdb) continue
$3 = "initialize"
$4 = [Movieclip 4198]
$5 = "RectBorder"
$6 = "initialize"
$7 = [Movieclip 4598]
$8 = "RectBorder"
$9 = "labelChanged"
$10 = [Movieclip 4868]
$11 = "Button"
$12 = "initialize"
$13 = [Movieclip 4868]
$14 = "Button"
$15 = "childCreated"
$16 = [Movieclip 4425]
...

```


CHAPTER 34

Profiling ActionScript

The Macromedia Flex ActionScript Profiler helps identify performance bottlenecks in your applications. It can show you where too many calls to a particular method might be occurring, or where an object's instantiation might be taking too long. The Flex ActionScript Profiler analyzes and logs information about your ActionScript calls and frame statistics in all your SWF files and MXML applications.

Contents

| | |
|--------------------------|-----|
| About profiling | 769 |
| About the Profiler | 770 |
| Using the Profiler | 770 |
| Analyzing data | 776 |
| Troubleshooting | 780 |

About profiling

A code profiler is most commonly used to measure the effects of object instantiation and method calls. To best take advantage of a profiler, you should also have a good understanding of your application's architecture, as well as the way your application interacts with external resources.

The ActionScript Profiler can examine the ActionScript in your Flex or Macromedia Flash applications so that you can identify bottlenecks. Although the Profiler cannot directly drill down into external resources such as Enterprise JavaBeans (EJBs), you can wrap calls to these resources in ActionScript functions and profile them to gauge the total time.

Before using the Profiler, you should define the most common use cases of your application and target the code paths of those use cases with the Profiler.

Macromedia recommends that you do not run the Profiler against your entire application. Rather, you should isolate sections of ActionScript code in your application and profile each section separately in the form of unit tests. If you try to profile an entire enterprise application, the Profiler might produce too much data.

About the Profiler

The ActionScript Profiler records the time that Flash Player takes to perform tasks in ActionScript. Most commonly, you use the Profiler to determine how long an ActionScript function or method takes to execute, how often it is called, and how much time is spent executing in its descendants. The Profiler can also show you the length of time ActionScript uses to instantiate objects.

In addition, the Profiler records frame data, which shows you how long Flash Player takes to render a frame for the client. This helps identify which objects might be taking too long to initialize, or whether there are bottlenecks due to heavy graphics use or poor coding.

The Profiler relies on the Flash Debug Player to collect profiler data and store the data in files on the client in a binary format.

Flex stores the Profiler's output data in the Profiler web application's */profiler_web_root/WEB-INF/ProfilerData* directory. The default location is */profiler/WEB-INF/ProfilerData*.

Using the Profiler

The Profiler web application runs on any operating system and application server that supports Flex. However, to generate statistics for your application, you must use a Windows client to request your application.

To use the Profiler:

1. Install the profiler.war application on your application server.
2. Configure the ActionScript Profiler in the client's mm.cfg file.
3. (Optional) Add `profile()` method calls to profile ActionScript blocks in MXML or limit the profiling in FLA files (for components). If you want to profile the entire application, do not add `profile()` methods.
4. Request your application and generate the SWD file. In most cases, you can request the application in a browser and append `?asprofile=true` to your request string. For example:
`http://localhost:8101/flex/myApplication.mxml?asprofile=true`
5. Request the ActionScript Profiler application to examine the results; for example:
`http://localhost:8101/profiler`

The following sections describe these steps in detail.

Installing profiler.war

When you install Flex, you usually create a directory to act as the context root for the Flex application on your Java application server. For example, on JRun, you create the following directory:

```
jrun_root/servers/server_name/flex
```

The default directory name is `/flex`. To use the Profiler, you create a profiler directory at the same level as the `/flex` directory in the root of your application server. The Profiler runs as a separate application from Flex. For example, on JRun, you create the following directory:

```
jrun_root/servers/server_name/profiler
```

The application directory structure should look similar to the following:

```
/app_server_root
  /flex/
  /flex/META-INF/..
  /flex/WEB-INF/..
  /profiler/
  /profiler/META-INF/..
  /profiler/WEB-INF/..
```

To install the Profiler:

1. Create the profiler's application root directory.
2. Find the profiler.war file that was installed by Flex. It is located in the Flex install folder. The default install folder in Windows is `C:/Program Files/Macromedia/Flex`.
3. Use WinZip, jar, or other archiving utility to extract the contents of the profiler.war file to the profiler application root that you created in step 1; for example, from the profiler's application root directory, type the following command:

```
c:/jrun4/servers/server1/profiler> jar -xvf "c:/Program Files/Macromedia/
flex/profiler.war"
```

Note: Macromedia recommends that you expand the contents of WAR files when you deploy them, rather than copy the entire WAR file itself into the application server's directory. By expanding the contents, you have greater visibility into the directory structure and contents of the WAR file.

4. (Optional) Start or restart your application server. After you deploy a new application, you might be required to restart your application server. For more information, see your web application server's documentation.

Configuring the Profiler

You configure the ActionScript Profiler by adding parameters to the `mm.cfg` text file. When you install Flex, Flex creates a file in the `flex_install_dir/bin` directory. You must copy the `mm.cfg` file to the client computer's home directory and edit its settings there.

The home directory is also the location of the `flashlog.txt` file which stores trace output, system errors, and warning messages generated by the Flash Debug Player while running a Flex application. Two Microsoft Windows environment variables define the location of the home directory:

HOMEDRIVE Specifies the drive letter of the path to the home directory. In most Microsoft Windows systems, the default value is `C:`, the primary hard drive.

HOMEPATH Specifies the path to the home directory, relative to **HOMEDRIVE**. On Microsoft Windows 2000 and Windows XP, the default is `/Documents and Settings/user_name` where `user_name` is your system user name.

To determine the current home directory on your Windows system, issue the following command at the command prompt:

```
echo %HOMEDRIVE%%HOMEPATH%
```

After you have copied the mm.cfg file to the home directory, you can edit the file to configure the profiler. When you change the mm.cfg file, you must close all browser windows for the changes to take effect.

The following example shows the default ActionScript Profiler settings from the mm.cfg file:

```
ProfilingOutputFileEnable=1
ProfilingOutputDirectory=c:\jrun4\servers\flex_server\profiler\WEB-INF\ProfilerData
FrameProfilingEnable=0
ProfileFunctionEnable=0
```

The following table describes the Profiler settings in the mm.cfg file:

| Parameter | Description |
|---------------------------|--|
| ProfilingOutputFileEnable | Set to 1 to enable profiling. Set to 0 to disable profiling. The default value is 0 (disabled). |
| ProfilingOutputDirectory | The full system path to the output directory for the Profiler *.dat file. Flex sets this path during installation. You must have write permission on this directory. |
| FrameProfilingEnable | Set to 1 to turn on frame-based profiling. Set to 0 to turn it off. The default value is 0. Caution: Frame-based profiling can generate a large amount of data if run continuously against a movie with many frames. When using the Profiler with an MXML application, frame-based profiling should be off, since the application is typically run in a single frame. For more information about frame-based profiling, see "Analyzing frame statistics" on page 779 . |
| ProfileFunctionEnable | Set to 1 to profile <i>only</i> ActionScript that is wrapped in the profile functions in your MXML applications. Set to 0 to profile <i>all</i> ActionScript calls in SWF files and MXML applications. The default value is 1. For more information about using profile functions to profile ActionScript, see "Adding profile methods to your ActionScript blocks" on page 773 . |

The Profiler settings in the mm.cfg file combine to provide a highly customizable environment for profiling your application. Two common settings are as follows:

- To turn off profiling, set `ProfilingOutputFileEnable` to 0.
- To generate the greatest amount of profiling data, set `ProfilingFileEnable` and `FrameProfilingEnable` to 1, and `ProfileFunctionEnable` to 0.

The following table describes the possible combinations of Profiling settings in the mm.cfg file:

| Description | ProfilingOutput FileEnable | FrameProfiling Enable | ProfileFunction Enable |
|--|-------------------------------|--------------------------|---------------------------|
| Generates no output. In all cases, if you set <code>ProfilingOutputFileEnable</code> to 0, no profiling data is generated. | 0 | N/A | N/A |
| Profiles only ActionScript wrapped in profile method calls (in either MXML applications or FLA files). | 1 | 0 | 1 |
| Profiles frame data and script blocks wrapped in profile method calls. These settings profile frame data for all SWF files and MXML applications, and profile ActionScript for MXML applications only. | 1 | 1 | 1 |
| Profiles frame data and all ActionScript for all SWF files and MXML applications. | 1 | 1 | 0 |
| Profiles all ActionScript for all SWF files and MXML applications. Does not generate frame data. | 1 | 0 | 0 |

Adding profile methods to your ActionScript blocks

To profile ActionScript blocks in your MXML and FLA files, you can wrap the contents of individual ActionScript functions in the profile method. To limit your profiling to just these methods, set the value of the `ProfileFunctionEnable` property to 1 in the mm.cfg file.

The profile method has the following signature:

```
profile(boolean);
```

To turn on profiling for the current function, call `profile(true)` inside the function or method definition. To turn off profiling, call `profile(false)` before the end of the function or method.

The following example of an MXML file ActionScript code block profiles the `setTime` method, but not the `getTime` method:

```
<mx:Script><![CDATA[
    function setTime() {
        profile(true);
        ...
        profile(false);
    }

    function getTime() {
        ...
    }
}]></mx:Script>
```

If you do not close the profile block, the Profiler continues data collection until one of the following occurs:

- You call `profile(false)`.
- Flash Player stops.

Generating SWD files

SWD files contain debugging and profiling information. When you generate a SWD file, Flash Player stores the SWD file in the same directory as the source file and the SWF file, with the same name as the SWF file (but a different file extension).

To profile MXML applications, you must generate a SWD file (Flash debug file) so that the profiler can access the appropriate hooks in your application. The SWD file and the SWF file must be synchronized. If you generate a new SWF file, you must generate a new SWD before running the profiler against your application.

You can generate a SWD file for an MXML file using the Debug version of the ActiveX Flash Player, the `mxmcl` command-line compiler, or using the Debug version of the stand-alone Flash Player. The client must be a Windows machine, but the server can be running on any supported platform. To profile a component SWC file that you wrote using the Flash authoring environment, you must export the file with debug settings enabled.

Regardless of how you want to generate profiling data, you must ensure that `<production-mode>` is set to `false` in the `flex-config.xml` file. If it is set to `true`, Flex does not generate a SWD file. The default value is `false`.

This section describes how to generate SWD files for MXML applications and for FLA files. For more information on using the `mxmcl` compiler, see [“Using the command-line compiler” on page 796](#).

Generating SWD files with the ActiveX Flash Player

You must be running Internet Explorer and have the latest `Flash.ocx` file installed to generate a SWD file using your browser. The latest version of this ActiveX control is installed in the `flex_install_dir/bin` directory during the Flex installation. Use the installation instructions included with Flex to install the ActiveX control.

To generate profiling data for an MXML application, perform one of the following tasks:

- Set the value of the `<generate-profile-swfs>` tag in the `flex-config.xml` file to `true`, and then request the MXML application in your browser; for example:

`http://localhost:8100/flex/myApp.mxml`

For more information on configuring Flex with the `flex-config.xml` file, see [“Editing compiler settings” on page 813](#).

- Request the MXML file in your web browser and append the `?asprofile=true` query string to the MXML application’s URL in your browser; for example:

`http://localhost:8100/flex/myApp.mxml?asprofile=true`

Appending `?asprofile=true` to your query string overrides the `<generate-profile-swfs>` setting in the `flex-config.xml` file and lets you selectively generate profiling data.

Note: Set `<production-mode>` to `false` to generate profiling data. You cannot generate a SWD file if `<production-mode>` is set to `true` in the `flex-config.xml` file. Furthermore, if production mode is enabled, you cannot override the `<generate-profile-swfs>` setting by appending `?asprofile=true` to your request string.

Once generated, the SWD file remains in memory. To save it on disk, set `<keep-generated-swfs>` to `true` in the `flex-config.xml` file. This is not required, but by writing the SWD file to disk, you can be sure that Flex is generating it properly.

Generating SWD files with the stand-alone player

To generate a SWD file in Windows or on UNIX using the stand-alone Flash Debug Player, open the SWF file using Flash Debug Player. This requires that first you generate a SWF file using either the mxmlc compiler or by requesting the MXML file in a browser. The stand-alone Flash Debug Player supports debugging and profiling.

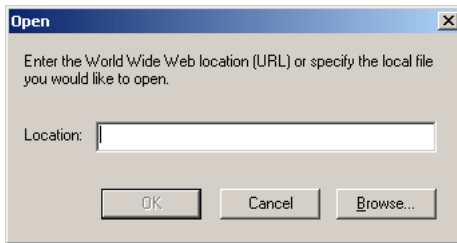
Flash Debug Player is located in `Flex_install_dir/bin/SAFlashPlayer.exe`.

For more information on using the mxmlc compiler, see [“Using the command-line compiler” on page 796](#).

To open a SWF file in Flash Debug Player:

1. Select File > Open.

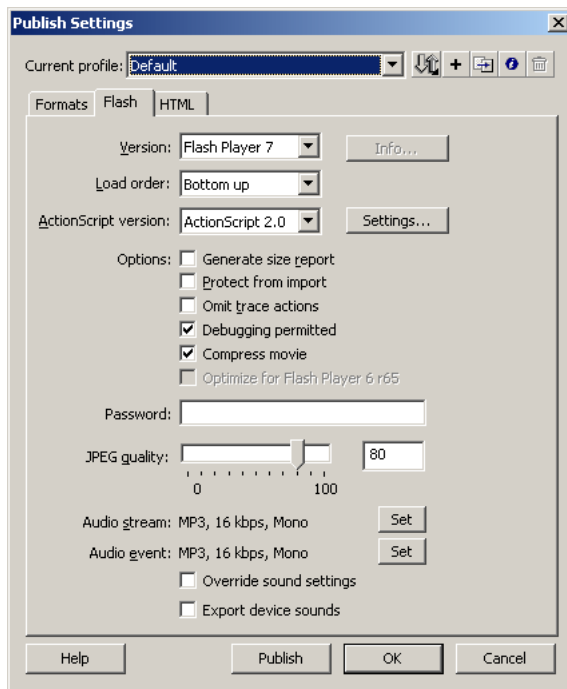
The Open dialog box appears:



2. Enter the path to the SWF file in the Location field.
3. Click OK.

Generating SWD files in the Flash authoring environment

To analyze SWF files generated from the Flash authoring environment, you must publish the Flash content with the Debugging Permitted setting selected in the Publish Settings dialog box, as the following figure shows:



When you generate a SWF file with Debugging Permitted selected, Flash generates a SWD file. Without the SWD file, Flash Debug Player runs the SWF file but does not collect performance data from the SWF file.

The SWF and SWD files share the same name with different extensions; you must store them in the same directory at runtime. If the SWD file is not found, Flash Debug Player throws a 404 (Not Found) error.

Analyzing data

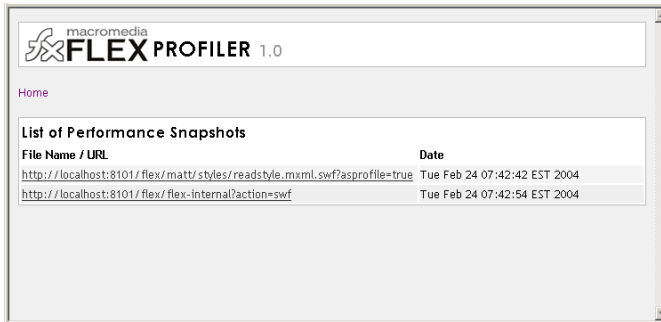
After you generate a SWD file for your application, run the ActionScript Profiler by opening the following URL:

`http://hostname:port/profiler`

For example:

`http://localhost:8101/profiler`

The Profiler main page appears:



The following table describes fields on the Profiler the main page:

| Field | Description |
|-----------|---|
| File Name | The name of the SWF file or MXML application that you ran the ActionScript Profiler against. Click a filename to view that application's Profiler data. |
| Date | The date that the Profiler ran against this file. |

To drill down into individual SWF file or MXML application profiling data, select the name of the application. You can sort most table data in the Profiler by clicking on the column header.

If the Profiler main page appears, but there are no performance snapshots, check the value of the `ProfilingOutputDirectory` setting in the `mm.cfg` file and ensure that it points to the proper directory. For more information about the `mm.cfg` settings, see [“Configuring the Profiler” on page 771](#).

The following sections describe the reports available in the ActionScript profiler.

Analyzing user-defined methods, functions, and modules

The User-Defined Methods report summarizes the calls to each custom ActionScript method, function, and module in your application.

You can view the source code for user-defined functions. If there are no custom functions called by the current application or method, the following message appears in place of the table:

"There are no user-defined methods, functions or modules"

When you drill down to a dependent function to see its self-time, the Profiler displays the time spent in that dependent function only.

The following table describes the fields on the User-Defined Methods report:

| Field | Description |
|---------------------------|---|
| Name | The names of the methods called during this Profiler run. Select the method name to see the Method Summary table for methods that were called by this method. |
| Calls | The number of times each method was called during this Profiler run. |
| Cumulative Time | The total amount of time it took to run this method, and all methods invoked within this method, excluding recursive calls to those child methods. |
| Cumulative Time (Average) | The average amount of cumulative time it took to run this method and all the methods invoked within this method. |
| Self Time | The amount of time just this method took to run. |
| Self Time (Average) | The average time this method took to run. Self Time (Average) is equal to the Self Time divided by the number of Calls. |

Analyzing built-in functions

The Built-in Functions report shows statistics for ActionScript methods, functions, and modules called by your ActionScript code. You cannot view the source code for built-in functions. If there are no built-in functions called by the current application or method, the following message appears in place of the table:

"There are no built-in functions"

The following table describes the fields on the Built-in Functions report:

| Field | Description |
|---------------------------|---|
| Name | The name of the built-in function. |
| Calls | The number of times each function was called during this Profiler run. |
| Cumulative Time | The total amount of time it took to run this function and all functions invoked within this function, excluding recursive calls to those child functions. |
| Cumulative Time (Average) | The average amount of cumulative time it took to run this function. The Cumulative Time (Average) is the Cumulative Time divided by the number of Calls. |

Analyzing source code

The Source Code report shows the ActionScript source code for the SWF file. The following table describes the fields in the Source Code report:

| Field | Description |
|-------|--|
| Line | The line number of the ActionScript source code. |
| Calls | The number of times this line was executed. |

| Field | Description |
|-----------------|---|
| Cumulative Time | The total time all calls took to execute this line. |
| Average Time | The average amount of time it took to execute this line. Average Time is equal to Cumulative Time divided by Calls. |
| Source | The ActionScript source code. |

Analyzing asynchronous function latencies

Asynchronous functions are functions that usually make network calls or calls to other functions that do not immediately return results. For example, calls to web services or Flash Remoting objects, or calls that use callback handlers are asynchronous function calls. In these examples, Flash Player continues to execute the next statement without waiting for the result of the asynchronous function call, so the Profiler separates out the data.

The Asynchronous Function Latency Summary shows the wait time (or “lag” time) before the client executes the function. The Profiler only displays function calls with registered handlers.

The following table describes the fields in the Asynchronous Function Latency Summary:

| Field | Description |
|-----------------|--|
| Name | The name of the function. |
| Calls | The number of calls made to this function during the data collection period. |
| Average Latency | The average amount of time the Flash client takes before it begins executing the function. |

Analyzing frame statistics

The Frame Statistics report shows profiling data for each frame rendered by Flash Player.

The following table describes the fields in the Frame Statistics report:

| Field | Description |
|---|---|
| Frame | The frame number that Flash Player renders for the current timing data. This is not the frame number in the FLA file of a multiframe SWF file. |
| Script Time Before Frame Start | The latency experienced before Flash Player renders the frame. Typically, Flash Player executes some ActionScript, such as initialization or determining the position and size of objects, before it renders a frame. |
| Script Time Between Frame Start and Frame End | The amount of time it takes to execute only the ActionScript while Flash Player is rendering this frame. |
| Frame Delay (Frame End – Frame Start) | The total amount of time it takes for Flash Player to render this frame. Frame Delay is the result of Frame End minus Frame Start. |

Troubleshooting

If Flex does not generate a SWD file or there is no *.dat file in the Profiler directory, check for the following:

- If you make changes to the mm.cfg file, you must close all browser windows for the changes to take effect.
- The *flex_app_root*/WEB-INF/flex/frameworks_debug/mx_debug.swc file must be present.
- The <debug-lib-path> setting in flex-config.xml file must point to *flex_app_root*/WEB-INF/flex/frameworks_debug/.
- The <production-mode> setting in the flex-config.xml file must be set to false to generate profiling data.
- The value of the ProfilingOutputFileEnable property is set to 1 in the mm.cfg file on the client requesting the application.
- Check that Flex is writing data files to the *server_root*/profiler/WEB-INF/ProfilerData directory. If the application to be profiled is small and you have minimal profiling information, Flex might not initially write a *.dat file. Close the browser running the application before running the profiler application. This flushes the buffer and forces Flex to write the data to disk.

CHAPTER 35

Using the Flex JSP Tag Library

Macromedia Flex includes a JSP tag library that you use to add MXML code to your JavaServer Pages (JSPs) or create custom HTML wrappers for your Flex applications. This chapter describes how to use the tag library.

Contents

| | |
|---|-----|
| Introduction to the Flex JSP tag library | 781 |
| Using the Flex JSP tag library. | 782 |
| About the Flex tags | 782 |
| Using the <code><mxml></code> tag | 785 |
| Using the <code><flash></code> tag | 788 |
| Using the <code><param></code> tag. | 789 |
| Using the <code><flashvar></code> tag. | 789 |

Introduction to the Flex JSP tag library

Flex includes a tag library that you can use with JavaServer Pages (JSPs). The Flex JSP tag library does the following:

- Supports inlined, dynamically generated MXML
- Generates an HTML wrapper for compiled MXML applications
- Generates an HTML wrapper for static SWF files

When you use the Flex JSP tag library, you have access to all of the objects on JSPs to dynamically generate the MXML from a combination of MXML and JSP tags.

For example, suppose you run a weather service that stores a ZIP code or geocode in your user's session object. Based on this geocode, you dynamically invoke a regional weather web service. Because you can write a Flex application in a JSP, you can access the session data and assign the proper web service before the MXML is compiled.

The Flex JSP tags are integrated with the following Flex features:

- Caching
- History management
- Accessibility
- Player deployment
- Player version detection

The Flex JSP tag library also generates the HTML wrapper around your Flex applications. As a result, you can use it to set `flashVars` or other properties of the `<object>` and `<embed>` tags.

Using the Flex JSP tag library

To use the Flex tag library, you add a `taglib` directive in your JSP that points to the `FlexTagLib` URI, as the following example shows:

```
<%@ taglib uri="FlexTagLib" prefix="mm" %>
```

As with any JSP `taglib` directive, this line must appear before you use any tags in the Flex JSP tag library.

The Flex JSP tag library is stored in the `flex-bootstrap.jar` file. This location is defined by the `taglib` declaration in the *flex_app_root*/WEB-INF/web.xml file. The default `taglib` declaration from the web.xml file is as follows:

```
<taglib>
  <taglib-uri>FlexTagLib</taglib-uri>
  <taglib-location>/WEB-INF/lib/flex-bootstrap.jar</taglib-location>
</taglib>
```

About the Flex tags

The Flex JSP tag library includes the following tags:

- `<mxml>` References an external MXML file or adds MXML tags inline in your JSP page.
- `<flash>` References a precompiled SWF file.
- `<param>` Passes predefined settings to Macromedia Flash Player.
- `<flashvar>` Passes `flashVars` variables to the SWF file.

The `<mxml>` and `<flash>` tags support HTML parameters as tag properties that define the appearance of the Flex application on the page.

When Flex receives a request for a JSP page that uses the `<mxml>` tag, it compiles the MXML file and dependent files into a SWF file, and then generates an HTML wrapper that references the new SWF file. For a JSP page that uses the `<flash>` tag, Flex generates the HTML wrapper and returns the precompiled SWF file.

About tag properties

The `<mxml>` and `<flash>` tags support a set of properties that define the presentation of the SWF file on the generated HTML page. These properties are defined by the HTTP specification. When you set a property, Flex adds that property to the `<object>` and `<embed>` tags within the HTML wrapper. In some cases, the properties are only supported by the `<object>` or the `<embed>` tag, but not both.

The following properties are supported by the `<mxml>` and `<flash>` JSP tags:

| | | | | | | | |
|---------|---------|--------|---------|---------|---------|--|-------------|
| align | archive | base | bgcolor | border | classid | codetype | data |
| declare | dir | height | hspace | id | lang | name | pluginspage |
| quality | salgn | scale | source | standby | style | supportembed (<code><flash></code> tag only) | tabindex |
| title | type | usemap | vspace | width | wmode | | |

For descriptions of these tags, see [“About the `<object>` and `<embed>` tag properties” on page 848](#).

For example, you can add a border to your SWF file by setting the value of the `border` property, as the following example shows:

```
<mm:mxml border="5" >
...
</mm:mxml>
```

Flex writes tag properties as parameters on the resulting HTML wrapper.

The `<mxml>` and `<flash>` JSP tags also take a set of intrinsic JavaScript events as properties. These events let the generated Flex application interact with the Document Object Model for the HTML page that contains it. The following table describes the JavaScript events that the `<mxml>` and `<flash>` JSP tags support:

| Attribute | Description |
|-------------------------|---|
| <code>onclick</code> | Occurs when the user clicks the mouse button over the Flex application's SWF file. The <code>onclick</code> event takes a value that is a script. The script executes whenever this event occurs for that SWF file. |
| <code>ondblclick</code> | Occurs when the user double-clicks the mouse button over the Flex application's SWF file. The <code>ondblclick</code> event takes a value that is a script. The script executes whenever this event occurs for that SWF file. |
| <code>onkeydown</code> | Occurs when the user presses a key down over the Flex application's SWF file. The <code>onkeydown</code> event takes a value that is a script. The script executes whenever this event occurs for that SWF file. |
| <code>onkeypress</code> | Occurs when the user presses and releases a key over the Flex application's SWF file. The <code>onkeypress</code> event takes a value that is a script. The script executes whenever this event occurs for that SWF file. |
| <code>onkeyup</code> | Occurs when the user releases a key over the Flex application's SWF file. The <code>onkeyup</code> event takes a value that is a script. The script executes whenever this event occurs for that SWF file. |

| Attribute | Description |
|--------------------------|--|
| <code>onmousedown</code> | Occurs when the user presses the mouse button over the Flex application's SWF file. The <code>onmousedown</code> event takes a value that is a script. The script executes whenever this event occurs for that SWF file. |
| <code>onmousemove</code> | Occurs when the user moves the mouse while the mouse pointer is over the Flex application's SWF file. The <code>onmousemove</code> event takes a value that is a script. The script executes whenever this event occurs for that SWF file. |
| <code>onmouseout</code> | Occurs when the user moves the mouse pointer away from the Flex application's SWF file. The <code>onmouseout</code> event takes a value that is a script. The script executes whenever this event occurs for that SWF file. |
| <code>onmouseover</code> | Occurs when the user moves the mouse pointer onto the Flex application's SWF file. The <code>onmouseover</code> event takes a value that is a script. The script executes whenever this event occurs for that SWF file. |
| <code>onmouseup</code> | Occurs when the user releases the mouse button over the Flex application's SWF file. The <code>onmouseup</code> event takes a value that is a script. The script executes whenever this event occurs for that SWF file. |

The following example defines a pair of JavaScript functions, and instructs the generated Flex application's SWF file to react to the user's mouse movements. When the user moves their mouse over the SWF file or off of the SWF file, the `message` field is updated, as in the following example:

```
<%@ taglib uri="FlexTagLib" prefix="mm" %>
<SCRIPT LANGUAGE="JavaScript">
    function showMessage() {
        message.value = "Use this application wisely";
    }

    function hideMessage() {
        message.value = "";
    }
</SCRIPT>

<mm:mxml border="5" onmouseover="showMessage();" onmouseout="hideMessage();"
...
</mm:mxml>
<TABLE>
    <TR>
        <TD><input type="text" name="message" size="50"></TD>
    </TR>
</TABLE>
```

About caching

When you use MXML inline on a JSP page, Flex returns a cached SWF file each time the JSP page is invoked if the MXML or the JSP has not changed. Otherwise, Flex recompiles a new SWF file and returns that. Even when you dynamically generate MXML in a CFM or JSP page, the generated source is cached in addition to the output SWF file. Flex caches the dynamically generated fragment of MXML, and associates the SWF file with this fragment.

Note: Compilation performance depends on the number of unique MXML fragments in your source JSP. Thus, you should use dynamic MXML sparingly.

If you are continuously generating new, unique pieces of MXML source code with the JSP, the cache might not be effective and might result in excessive recompilation. When the MXML source is in its own file, the source is not dynamic and, therefore, only the output SWF file is cached. In general, you should only use dynamic MXML if you can assure yourself that the number of unique pieces of MXML source is finite and roughly less than the size of the MXML source cache.

You can set the number of fragments that Flex caches in the `flex-config.xml` file. For more information, see [“Configuring caching” on page 808](#).

Using the `<mxml>` tag

You use the `<mxml>` JSP tag to write Flex applications inside a JSP. This tag supports adding MXML content in the JSP itself, or refers to external MXML files that Flex compiles and embeds in the JSP. The syntax for the `<mxml>` tag is as follows:

```
<prefix:mxml [source=source_file] [properties]>
    [MXML]
</prefix:mxml>
```

The `<mxml>` tag creates the `<object>` and `<embed>` tags that act as an HTML wrapper for the Flex application. This wrapper is the same as the wrapper generated when you request a `*.mxml` file.

You can define attributes of the `<mxml>` tag that become properties of the `<object>` and `<embed>` tags in the HTML wrapper. These properties define how the SWF file appears and interacts with the web page. For information about using the `<mxml>` tag to define the values in the HTML wrapper, see [“About the `<object>` and `<embed>` tags” on page 846](#).

The `<mxml>` tag has optional child tags: `<param>` and `<flashvar>`. You use the `<param>` tag to pass predefined variables to Flash Player. You use the `<flashvar>` tag to pass user-defined variables. For more information, see [“Using the `<param>` tag” on page 789](#) and [“Using the `<flashvar>` tag” on page 789](#).

The following sections describe how to use the `<mxml>` JSP tag to write MXML directly in a JSP and how to use it to refer to an external MXML document.

Writing MXML in JSPs

You use the `<mxml>` tag without a `source` property to enclose a set of MXML tags in a JSP page. You can set any number of properties that define the presentation of the SWF file on the page in the `<mxml>` tag. The syntax for writing MXML inline in a JSP is as follows:

```
<prefix:mxml [properties]>
    <mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
        //MXML tags
    </mx:Application>
</prefix:mxml>
```

When you write MXML directly in a JSP, you set the prefix in the `<mx:Application>` tag just as you would when writing any Flex application.

The following example creates a Flex application with an Accordion container in the JSP file:

```
<%@ taglib uri="FlexTagLib" prefix="mm" %>
<html><body>
<h3>Introduction</h3>
<p>This is an example of writing MXML in a JSP.</p>
<h3>My App</h3>
<mm:mxml border="1">
  <mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
    <mx:Accordion width="500" height="300">
      <mx:VBox label="panel1" width="500" height="200" />
    </mx:Accordion>
  </mx:Application>
</mm:mxml>
</body></html>
```

You can write multiple Flex applications in a single JSP. The Flex applications do not share the same namespace and run independently of one another. Thus, you can use the same variable names and control IDs across applications without causing conflicts.

The following example shows two identical applications defined inline in a JSP:

```
<%@ taglib uri="FlexTagLib" prefix="mm" %>
<html><body>
<h3>Introduction</h3>
<p>This is an example of writing multiple MXML apps in a single JSP.</p>
<h3>My App 1</h3>
<mm:mxml border="1">
  <mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
    <mx:Accordion width="500" height="300">
      <mx:VBox label="panel1" width="500" height="200" />
    </mx:Accordion>
  </mx:Application>
</mm:mxml>
<HR>
<h3>My App 2</h3>
<mm:mxml border="1">
  <mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
    <mx:Accordion width="500" height="300">
      <mx:VBox label="panel1" width="500" height="200" />
    </mx:Accordion>
  </mx:Application>
</mm:mxml>
</body></html>
```

Mixing JSP expressions with MXML

When you write MXML code in a JSP, you can mix JSP expressions with MXML code to dynamically generate Flex applications. The following simple example uses a call to `session.getProperty()` to get the user's name out of the JSP's session object, and uses that name as the value of the label's text property in the MXML:

```
<%@ taglib uri="FlexTagLib" prefix="mm" %>
<mm:mxml>
  <mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" width="200"
    height="240">
    <mx:Label id="label0" text="Hi <%= session.getAttribute("username") %>" />
  </mx:Application>
</mm:mxml>
```

Macromedia recommends that you do not use per-request data in the MXML, because every time the data changes, Flex recompiles the MXML tags into a new SWF file. In the previous example, Flex recompiles whenever a new user requests the page, but does not recompile if the same user requests the page multiple times. If the fragment does not change, recompilation is not necessary.

You can use any JSP expression within the MXML code, as long as it evaluates to a String. The following example gets a current counter and loads a picture indexed by the counter when running the SWF file. If the user clicks the Refresh button in the browser, the counter is incremented and the next image in the array is displayed, as the following example shows:

```
<%@ taglib uri="FlexTagLib" prefix="mm" %>
<%
  int counter=0;
  try {
    counter=Integer.parseInt((String)application.getAttribute("counter"));
  } catch (Exception e) {
  }
  String jpgs[]={"mike.jpg","steve.jpg","matt.jpg"};
  String names[]={"Mike","Steve","Matt"};
%>
<mm:mxml>
  <mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" width="300"
    height="200" >
    <mx:Label text="<%=names[counter]%>" />
    <mx:Image width="150" height="75" source="<%=jpgs[counter]%>" />
  </mx:Application>
</mm:mxml>
<%
  counter=(counter+1)%4;
  application.setAttribute("counter",""+counter);
%>
```

Including external MXML files in JSPs

You use the `<mxml>` tag with the `source` property to include external MXML files in your JSPs. You can include multiple Flex applications on a single page and dynamically select which MXML application to run on the page. Flex generates a new SWF file for each source file, and it is a separate MXML application.

The syntax for including an external MXML file with the `<mxml>` tag is as follows:

```
<prefix:mxml source="path_to_MXML_file"/>
```

The `source` property is relative to the location of the JSP.

When you point to external MXML files, the custom tag only recompiles the MXML application into a SWF file if the source files have changed.

You can include any number of applications in a single JSP using the `<mxml>` tag. The following example embeds two applications in a JSP:

```
<%@ taglib uri="FlexTagLib" prefix="mm" %>
<h3>First App</h3>
<mm:mxml source="../FlexApps/MyApp.mxml" />

<h3>Second App</h3>
<mm:mxml source="../FlexApps/MyApp2.mxml" />
```

Using the `<flash>` tag

You use the `<flash>` tag to include pregenerated SWF files in your JSPs. The `<flash>` tag generates the HTML wrapper that defines the application on the page. You can set tag attributes on the `<flash>` tag that Flex converts to properties that define the appearance and interaction of the SWF file on the page.

To use the `<flash>` tag, you must first generate the SWF file by requesting it with Macromedia Flash Player or browser, or the mxmmlc precompiler.

The syntax of the `<flash>` tag is as follows:

```
<prefix:flash source=source.swf [attributes] />
```

For example:

```
<%@ taglib uri="FlexTagLib" prefix="mm" %>
<mm:flash source="myApp.swf" border="5" />
```

The `<flash>` tag creates the `<object>` and `<embed>` tags that act as an HTML wrapper for the Flex application and adds them to the resulting HTML output stream.

The `<flash>` tag has optional child tags: `<param>` and `<flashvar>`. You use the `<param>` tag to pass predefined variables to Flash Player. You use the `<flashvar>` tag to pass user-defined variables. For more information, see [“Using the `<param>` tag” on page 789](#) and [“Using the `<flashvar>` tag” on page 789](#).

You can define attributes of the `<flash>` tag that become properties of the `<object>` and `<embed>` tags in the HTML wrapper. These properties define how the SWF file appears and interacts with the web page. For information on using the `<flash>` tag to define the values in the HTML wrapper, see [“About the `<object>` and `<embed>` tags” on page 846](#). You can also define these properties using the `<param>` tag.

When you use the `<flash>` JSP tag, Flex generates a new HTML wrapper every time the JSP is requested. There is no noticeable performance penalty for regenerating this wrapper, however, because the underlying SWF file is not recompiled. After the JSP page is generated, it is probably cached by your JSP engine as a servlet, as any JSP page would be cached. For more information, see your application server's documentation.

Using the `<param>` tag

You use the `<param>` JSP tag to set predefined properties of the Flash SWF file and `<object>` and `<embed>` parameters, such as the background color.

The syntax for the `<param>` tag is as follows:

```
<prefix:mxml | flash>
  ...
  <prefix:param name="param_name" value="param_value" />
  ...
</prefix:mxml | flash>
```

For a complete list of the display parameters you use to define the Flash SWF file, see [“About the `<object>` and `<embed>` tag properties” on page 848](#).

To pass dynamic data or user-defined variables to your Flex applications, use the `<flashvar>` tag to define the value of the `flashVars` variables. For more information about `flashVars`, see [“Using the `<flashvar>` tag” on page 789](#).

Using the `<flashvar>` tag

You use the `<flashvar>` child tag to pass user-defined variables into a Flex application. The tag takes two attributes, `name` and `value`, and can be a child tag of either the `<flash>` or `<mxml>` tags.

The following example passes in the properties `firstname` and `lastname` to the `test.swf` file:

```
<mm:flash source="test.swf">
  <mm:flashvar name="firstname" value="Nick" />
  <mm:flashvar name="lastname" value="Danger" />
</mm:flash>
```

Flex converts `<flashvar>` variables into parameters for the `<object>` and `<embed>` tags in the resulting HTML wrapper. Using the previous example, Flex renders the following HTML wrapper:

```
<object codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/
swflash.cab#version=7,0,14,0" classid="clsid:D27CDB6E-AE6D-11cf-96B8-
444553540000" width="700" height="300">
  <param name="movie" value="/dev/demos/test.swf">
  <param name='flashVars' value='firstname=Nick&lastname=Danger'>
    <embed type="application/x-shockwave-flash" src="/dev/demos/test.swf
" pluginspage="http://www.macromedia.com/go/getflashplayer" width="700"
height="300" movie="/dev/demos/test.swf"
flashVars='firstname=Nick&lastname=Danger'>
  </embed>
</object>
```

When you pass a value into a Flex application with a `<flashvar>` tag, you can then use the value of that variable in the Flex application, as long as the variable is declared inside an MXML script block.

To use the variable in your Flex application, declare the variable name but do not initialize it. Flex can then access the value in its global scope. The following JSP fragment sets the value of a `userID` in the body of the JSP, declares the variable name in the Flex application, and then uses it in a function:

```
<%@ taglib uri="FlexTagLib" prefix="mm" %>
<% String userID = "2405"; %>

<mm:mxml>
  <mm:flashvar name="userID" value="<%= userID %%" />
  <mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
    <mx:Script>
      var userID:String;
      function getUserID() {
        vb1.label = vb1.label + userID;
      }
    </mx:Script>
    <mx:Accordion>
      <mx:VBox id="vb1" label="Panel 1: ">
        <mx:Button label="Get User ID" click="getUserID();" />
      </mx:VBox>
    </mx:Accordion>
  </mx:Application>
</mm:mxml>
```

The value of a `<flashvar>` tag does not have to be static. It can be any JSP expression that can be evaluated to a `String`, as the following example shows:

```
<mm:flashvar name="userID" value="<%= x.toString(); %%" />
```

You can bind the value of a parameter using the curly braces (`{}`) shorthand, just as you would use curly braces in other instances. The following example prints the value that a parameter passes in a `<flashvar>` tag in the label:

```
<%@ taglib uri="FlexTagLib" prefix="mm" %>
<html><body>
<%
  session.setAttribute("username", "nick");
  String s = (String) session.getAttribute("username");
%>
<h3>Introduction</h3>
<p>This is an example of passing variables to an MXML application in a JSP.</p>
<h3>My App 2</h3>

<mm:mxml border="1">
  <mm:flashvar name="param1" value="<%=s%%" />
  <mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml">
    <mx:Script>
      var param1;
    </mx:Script>
    <mx:Label text="Hello {param1}" />
  </mx:Application>
</mm:mxml>
```

```
    </mx:Application>  
</mx:mxml>  
</body></html>
```

Since the value of `<flashvar>` is not part of the MXML fragment itself, Flex does not recompile the application when the value changes (although the application server may recompile the JSP page). This is a more efficient way of passing dynamic data to the Flex application than mixing JSP expressions inside the MXML code, but it restricts you to String values of a relatively short length. Also, your MXML code must declare a variable for each passed `<flashvar>` that you access inside the application.

In addition to passing variables into Flex applications with the `<flashvar>` tag, you can append them as query string parameters. Flex converts query string parameters to `flashVars` in the HTML wrapper. For more information, see [“Passing request data to Flex applications” on page 859](#).

PART V

Administering Applications

This part describes how to administrate Macromedia Flex applications.

The following chapters are included:

| | |
|--|-----|
| Chapter 36: Administering Flex | 795 |
| Chapter 37: Applying Flex Security | 823 |
| Chapter 38: Deploying Applications | 835 |

CHAPTER 36

Administering Flex

This chapter describes how to use the command-line MXML compiler and the Macromedia Flex application logger. It also discusses how to configure the Flex application settings using the `flex-config.xml` and `web.xml` files.

Contents

| | |
|---|-----|
| Overview | 795 |
| Using the command-line compiler | 796 |
| Using SWC files. | 800 |
| Editing the <code>flex-config.xml</code> file. | 806 |
| Changing application server settings | 817 |
| Configuring logging. | 818 |

Overview

You administer Flex with a combination of configuration file settings and application server settings. Flex runs as an application on your J2EE server, therefore, many settings such as security, access control, and virtual directory mapping are done using the web application server's configuration utilities rather than using the Flex configuration files.

The following table describes the most common configuration files that you use to configure the Flex environment:

| File | Description |
|---|--|
| flex_app_root/WEB-INF/web.xml | Configures your Flex application to run on the J2EE web application server. You use this file to define context parameters, filters, servlet mappings, JSP tag libraries, error handling, and other settings for your web application. You should not edit the web.xml file unless you are adding additional functionality to your web application or you are configuring Flex to run on a Macromedia ColdFusion server. For more information, see the Flex installation instructions. |
| flex_app_root/WEB-INF/flex/flex-config.xml | Configures Flex. You use this file to define debugging, compiler, cache, proxy, logging, font and other settings for Flex. This is the file that you most often edit to change the behavior of the Flex application running on your application server. |
| flex_app_root/WEB-INF/flex/gateway-config.xml | Configures the Macromedia Flash Remoting gateway. You can configure service adapters, security, logging, and other settings for Flash Remoting using this file. |

This chapter includes the following topics:

- Performing off-line compilation using the command line compiler. This mxmcl utility compiles MXML files into SWF files.
- Compiling components using the compc utility. This tool compiles SWC files from MXML or ActionScript class files.
- Setting configuration options in the flex-config.xml file. You use this file to define compiler, debugging, proxy, and other settings for Flex.
- Configuring Flex logging.

Using the command-line compiler

You can use the mxmcl utility included with Flex to compile your MXML files into SWF files without requesting them from a browser or Macromedia Flash Player. You use the utility to precompile Flex applications that you want to deploy on another server or to automate compilation in a testing environment.

Note: If you precompile a Flex SWF file, you can deploy it only on a server that is running a licensed copy of Flex.

To use mxmcl, you must have a Java runtime in your system path.

The syntax of the mxmcl utility is as follows:

```
mxmcl [options] filename.xml
```

To use the mxmmlc utility:

1. Open a console window.
2. Change to the Flex_*install_dir*/bin; for example, C:/Program Files/Macromedia/Flex/bin.
3. Invoke mxmmlc with the appropriate arguments.

In Windows systems, you can invoke the mxmmlc.exe executable. On all systems, you can invoke the mxmmlc batch file. In both situations, the syntax is the same; for example:

```
mxmmlc -configuration WEB-INF/flex/flex-config.xml myApp.mxml
```

Specify the `configuration` option to use the settings in the flex-config.xml file for compiling. If you specify both a `configuration` option and use other command-line options, the command-line options override the configuration file values where possible. If you specify the `configuration` or `flexlib` options, mxmmlc uses settings in the flex-config.xml file when they are not overridden by options on the command line.

For more information on using the flex-config.xml file, see [“Editing the flex-config.xml file” on page 806](#).

The following table describes the mxmmlc options:

| Option | Description |
|--|--|
| -accessible | Enables accessibility features when compiling the Flex application. The default value is <code>disabled</code> . For more information about creating accessible Flex applications, see “Enabling accessibility” on page 816 . |
| -aspath <i>path</i> | Adds directories or files to the ActionScript classpath. You can use wildcards to include all files and subdirectories of a directory. The default value is <code>flex_app_root/WEB-INF/flex/user_classes</code> . For more information on the ActionScript classpath, see “Editing the ActionScript classpath” on page 814 . |
| -batch <i>file1.mxml</i> [<i>file2.mxml</i> , ...] | Compiles multiple MXML files. Separate each filename with a space. When you use the <code>batch</code> option, mxmmlc outputs error and informational messages to <i>filename.err</i> and <i>filename.out</i> for each MXML file, rather than the console. These files are in addition to the <code>manager.err</code> and <code>manager.out</code> files. |
| -configuration <i>path</i> | Designates a full path or a path that is relative to the web application root that points to the configuration file. Flex includes a default configuration file, <code>flex-config.xml</code> , in the <code>flex_app_root/WEB-INF/flex</code> directory. If you specify a configuration file, you can override individual options by setting them on the command line. If you use the <code>configuration</code> option, then you must specify absolute paths in the flex-config.xml file. |
| -contextroot <i>root</i> | Specifies the value of the <code>@ContextRoot</code> directive in your MXML applications. You use this directive to make your applications more portable. |
| -debugpassword <i>password</i> | Lets you engage in remote debugging sessions with the Flash IDE. For more information, see “About the debugger” on page 752 . |

| Option | Description |
|--|---|
| <code>-encoding <i>encoding</i></code> | Specifies the encoding for the compiler. If there is no byte order mark and this option is not specified, the compiler uses the encoding in <code>flex-config.xml</code> . If there is no encoding defined in <code>flex-config.xml</code> , the compiler uses the platform default. |
| <code>-flexlib <i>path</i></code> | Specifies a directory containing frameworks and system_classes. This then leads mxmhc to the web application root for determining configuration options. |
| <code>-g</code> | Generates a SWD file for use by the ActionScript debugger. By default, mxmhc does not generate debugger information. |
| <code>-gatewayurl <i>url</i></code> | Specifies the callback URL for the Flash Remoting gateway to use when using AMF encoding over HTTP. This value is the equivalent of the <code><amf-gateway></code> element in the <code><remote-objects></code> block in the <code>flex-config.xml</code> file. For more information, see “Declaring a data service” on page 658 . |
| <code>-gatewayhttpsurl <i>url</i></code> | Specifies the callback URL for the Flash Remoting gateway to use when using AMF encoding over HTTPS. This value is the equivalent of the <code><amf-https-gateway></code> element in the <code><remote-objects></code> block in the <code>flex-config.xml</code> file. For more information, see “Declaring a data service” on page 658 . |
| <code>-genlibdir <i>dir</i></code> | Overrides the location of the directory in which Flex creates RSL SWC files. When invoked at runtime, Flex creates an RSL directory with the following path: <code>app_root/WEB-INF/flex/generated/libs/hash_value/</code> You override this directory because generating a SWF offline with mxmhc precludes the use of an application root. |
| <code>-headless</code> | Enables the headless implementation of the Flex compiler. This sets the following: <code>System.setProperty("java.awt.headless", "true")</code> The headless setting (<code>java.awt.headless=true</code>) is required to use fonts and SVG on UNIX systems without X Windows. |
| <code>-libpath <i>path</i></code> | The path to the base Flex classes and SWC files. The default is <code>flex_app_root/WEB-INF/flex/frameworks</code> . |
| <code>-loglevel <i>error warn info debug</i></code> | Sets the log level for the current compilation. Valid values are <code>error</code> , <code>warn</code> , <code>info</code> , and <code>debug</code> . For more information on log levels, see “Configuring logging” on page 818 . |
| <code>-namespace <i>uri</i> <i>manifestfile</i></code> | Specifies a namespace for the MXML file. You must include a URI and the location of the manifest file that defines the contents of this namespace. This path is relative to the MXML file. For more information about manifest files, see “Using manifest files” on page 805 . |
| <code>-o <i>path</i></code> | Specifies the output path and filename for the resulting SWF file. By default, mxmhc uses <code>mxml_filename.swf</code> in the current directory. This option has no equivalent compiler setting in the <code>flex-config.xml</code> file. |

| Option | Description |
|----------------------------|--|
| -O[0] | <p>Enables the ActionScript optimizer. Add the O (zero) to disable the ActionScript optimizer. The default is enabled.</p> <p>For more information about the ActionScript optimizer, see “Using the ActionScript optimizer” on page 813.</p> |
| -profile | <p>Generates a SWD file for use by the ActionScript Profiler. By default, mxmcl does not generate Profiler information.</p> |
| -proxyurl <i>url</i> | <p>Specifies the callback URL for the Flex proxy over HTTP.</p> <p>This value is the equivalent of the <url> elements in the <web-service-proxy> and <http-service-proxy> blocks in the flex-config.xml file. If you specify the proxyurl option on the command line, mxmcl overrides both of these settings.</p> <p>For more information, see “Declaring a data service” on page 658.</p> |
| -proxyhttpsurl <i>url</i> | <p>Specifies the callback URL for the Flex proxy over HTTPS.</p> <p>This value is the equivalent of the <https-url> elements in the <web-service-proxy> and <http-service-proxy> blocks in the flex-config.xml file. If you specify the proxyhttpsurl option on the command line, mxmcl overrides both of these settings.</p> <p>For more information, see “Declaring a data service” on page 658.</p> |
| -proxyallowurloverride | <p>Lets users override the proxyurl option by using a proxy specified with a flashVars variable or by appending the query string parameter ?proxyURL=url to the query string.</p> <p>The default is disabled. For more information, see “Declaring a data service” on page 658.</p> |
| -remoteallowurloverride | <p>Lets users override the remoteurl setting by using a proxy specified with a flashVars variable or by appending the query string parameter ?remoteURL=url to the query string.</p> <p>The default is disabled. For more information, see “Declaring a data service” on page 658.</p> |
| -report | <p>Generates the <i>filename</i>-report.xml file that lists configuration options and symbols used in the compilation. The file is located in the same directory as the source files.</p> |
| -systemclasses <i>path</i> | <p>Path to the native (intrinsic) ActionScript classes. The default is <i>flex_app_root</i>/WEB-INF/flex/system_classes.</p> |
| -version | <p>Returns the version number of the MXML compiler. If you are using a trial or Beta version of Flex, the version option also returns the number of days remaining in the trial period and the expiration date.</p> |
| -webroot <i>directory</i> | <p>Specifies the root of the web application. If you do not specify a webroot, mxmcl searches for a WEB-INF directory relative to the MXML document.</p> <p>This is the parent directory of the target application if the application is outside of a web application. This is the true web root if the application is in a web application.</p> |

The mxmcl compiler creates manager.err and manager.out files with error and informational messages about the file you compiled. If you compile more than one MXML file, these messages apply to all compiled files.

You commonly use the `webroot` and `configuration` options together. The `configuration` option specifies a `flex-config.xml` to use. The default `flex-config.xml` contains all relative paths to the web application. For example, the `actionscript-classpath` in the `flex-config.xml` file is often a relative path:

```
<actionscript-classpath>
  <path-element>/WEB-INF/flex/user_classes</path-element>
</actionscript-classpath>
```

You use the `webroot` option to set the directory that `mxmmlc` inserts to resolve relative paths. By using the `webroot` option, you do not have to edit the `flex-config.xml` and change all relatives paths to absolute paths; for example:

```
$ mxmmlc -webroot c:/JRun4/servers/default/flex -configuration
my-flex.config.xml MyApp.mxml
```

Using SWC files

A *SWC* file is an archive file for Flex components. SWC files make it easy to exchange components among Flex developers. You need only exchange a single file, rather than the MXML or ActionScript files and images and other resource files. In addition, the SWF file inside a SWC file is compiled, which means that the code is obfuscated from casual view. Finally, compiling a component as a SWC file can make namespace allocation an easier process.

SWC files can contain one or more components and are packaged and expanded with the PKZip archive format. You can open and examine a SWC file using WinZip, jar, or other archiving tool. However, you should not manually change the contents of a SWC file, and you should not try to run the SWF file that is in a SWC file outside of a SWC file.

The following table describes the contents of a SWC file:

| File | Description |
|------------------------|---|
| catalog.xml | Lists the contents of the component package and its individual components, and serves as a directory to the other files in the SWC file. |
| Source code | Contains one or more ActionScript files that contain class declarations for the components. The source code is used only for type-checking when extending components, and is not compiled by the authoring tool because the compiled bytecode is already in the implementing SWF file. The source code might contain intrinsic class definitions that contain no function bodies and are provided only for type-checking. |
| Implementing SWF files | Implements the components. One or more components can be defined in a single SWF file. |
| Live Preview SWF files | Supports Live Preview in the authoring tool. If omitted, the implementing SWF files are used for Live Preview instead. You can omit the Live Preview SWF file in nearly all cases; include the file only if the component's appearance depends on dynamic data (for example, a text box that shows the result of a web service call). |
| Debug info | Includes a SWD file corresponding to the implementing SWF file. The filename is always the same as that of the SWF file, but with the extension <code>.swd</code> . If it is included in the SWC file, debugging of the component is allowed. |

| File | Description |
|--------------------|---|
| Icon | Contains the 18 x 18, 8-bit-per-pixel icon used to display a component in the authoring tool user interface. If you do not supply an icon, a default icon appears. The icon must be a PNG file. |
| Property Inspector | Supports a custom Property Inspector in the authoring tool. If omitted, the default Property Inspector is displayed to the user. |

Flex includes a single SWC file that contains all the built-in components. This SWC file is located in the *flex_app_root*/WEB-INF/flex/frameworks directory.

Creating SWC files

To create a SWC file, use the `compc` utility in the *flex_install_dir/bin* directory. The `compc` utility generates a SWC file from MXML component source files and/or ActionScript component source files. You cannot use the command-line component compiler to create a SWC file from a FLA file or other file created in the Macromedia Flash MX authoring environment.

The syntax for using `compc` is as follows:

```
compc -root root [options] source_filename|source_directory
```

If you do not specify `root`, `compc` checks the values of the `flexlib` and `configuration` options for a valid application root. If you specify none of these, `compc` returns an error.

If you provide multiple source files, `compc` generates a single SWC containing those components. If you specify a directory, `compc` generates a single SWC file for all ActionScript and MXML files in that directory. In both cases, when `compc` creates a SWC file for multiple components, the SWC file is named `Merged.swc`.

The default name for the output SWC file is the same name as the component source file, but with a SWC extension. If you provide multiple source files, `compc` names the output SWC file after the first component it encountered during compilation.

In many ways, the `compc` command-line utility for creating SWC files is similar to the MXML command-line compiler utility, `mxmmlc` (see [“Using the command-line compiler” on page 796](#)). The `mxmmlc` utility produces a SWF file from an MXML file; the `compc` utility produces a SWC file from its input files. SWC files contain a SWF, plus ASI files that provide interfaces to the SWF file. Because `compc` and `mxmmlc` both produce SWF files, `compc` supports most of the `mxmmlc` command-line options such as `systemclasses`, `libpath`, and `proxyurl`. Like `mxmmlc`, you can point `compc` to the `flex-config.xml` configuration file for values rather than specify command line options, such as `proxyurl`.

The following table describes the `compc` options that are not used by `mxmmlc`:

| Option | Description |
|---------------------------------|--|
| <code>-manifest filename</code> | <p>Generates a text file that defines the component's namespace with a unique identifier (URI). You point to this file by adding a new <code><namespace></code> element in the <code>flex-config.xml</code> file.</p> <p>Manifest files are useful if you are generating a SWC file that contains components from more than one package.</p> <p>The MXML manifest file is located at <code>flex_app_root/WEB-INF/flex/mxml-manifest.xml</code>.</p> <p>For more information on using manifest files, see “Using manifest files” on page 805.</p> |
| <code>-o path</code> | <p>Specifies the output path and filename for the resulting SWC file. The default is the directory that the component source file is in; if there are multiple components, the default is the name of the first component.</p> <p>This option has no equivalent compiler setting in the <code>flex-config.xml</code> file.</p> |
| <code>-root root</code> | <p>Specifies the root of the location of the component. This is not the same as the web application root, although in most cases it is the same location.</p> <p>You usually specify the <code>root</code> option when the component is in a package. The value of <code>root</code> is a relative location based on whether that component is in a package or has a longer namespace.</p> <p>If you do not specify <code>root</code>, <code>compc</code> checks the values of the <code>webroot</code>, <code>flexlib</code>, and <code>configuration</code> options for a valid application root. If you specify none of these, <code>compc</code> returns an error.</p> <p>If all source files are not under the root, <code>compc</code> returns an error. If the package structures of the ActionScript files do not match the root, <code>compc</code> returns an error.</p> |

If the `flexlib` and `configuration` parameters are not in the same package structure as the first source file, then `compc` uses the directory of the first source file as the value of `root`. For example, when `flex` is located in the following directory:

```
C:\Flex
```

And the files you want to compile are located in the following directory:

```
C:\skinning
```

You can issue the following command:

```
compc -o MyTheme.swc MySampleRectBorder.as
```

The `compc` utility assumes in this case that the value of `flexlib` is `C:\Flex\lib`. `compc` then checks if `C:\Flex` would be the appropriate value for the `root` option. It is not, because `MySampleRectBorder.as` is not located in the same directory structure. As a result, `compc` uses `C:\skinning` as the `root` instead.

For more examples, see [“Examples using compc” on page 803](#).

The following mxmmlc options cannot be used with compc:

- accessible
- batch
- debugpassword

Examples using compc

The following example converts a custom component called MyButton.mxml into MyButton.swc. In this example, the application root (/flex) is one directory up from the current directory, the MXML file is in the current directory, and compc outputs a SWC file in a parallel directory named components.

```
C:\JRun4\servers\default\flex\workDir>compc -o ../components/MyButton.swc  
-root .. MyButton.mxml
```

When you compile a SWC file, you should store the new file in a location that is not the same as the component's source files. If both sets of files are accessible by Flex, unexpected behavior can occur.

You can set some compiler options at the application compiler level as well as at the component level. If you use a remote URL in your component, for example, you can embed that within the SWC file by using the `remoteurl` option for compc. Otherwise, the compiler will use the default value specified by the `<url>` element in the `<remote-objects>` block in the `flex-config.xml` file.

The following example specifies a remote URL of `remoteurl.com` for the MyComp component:

```
/server/flex>compc -o WEB-INF/flex/user_classes/MyComp.swc -root .  
-remoteurl http://remoteurl.com MyComp.mxml
```

If you then use mxmmlc to compile the application that references your new component, the `remoteurl` remains intact and is not affected by other `remoteurl` settings in the `flex-config.xml` file or in other components or source files. However, if you do not compile your component into a SWC file, but use it as an uncompiled ActionScript file, using the `remoteurl` option with mxmmlc applies to your component as well as the application files.

Compiling RSL SWC files

You can also use compc to compile runtime shared library (RSL) SWC files from the library descriptor file. In this case, you specify the SWS file rather than the MXML file as an argument to compc, as the following example shows:

```
compc myLib.sws
```

The compc compiler outputs a SWC file that you can then specify as the RSL source in your applications. The default output directory is `flex_app_root/WEB-INF/flex/generated/hash-map/library_name.swc` file. You can override this directory using the `genlibdir` option on the command line.

For more information on using RSLs, see [Chapter 25, “Using Runtime Shared Libraries,” on page 595](#).

Working with namespaces

You usually create components within a package or a directory structure that separates them from the application code. This logical separation of components promotes reuse and helps organize what can become a large fileset for a single application. In some cases, SWC files contain components that are from multiple packages.

When you deploy packaged components, they must have a unique identifier for a namespace so that Flex can locate the component within the SWC file. However, when you create a component that resides at the top-level directory of a Flex application, you can use a generic namespace.

Using top-level components

The following `comp` call creates a SWC file from the `MyOtherButton.mxml` component file and stores it in the `flex_app_root/WEB-INF/flex/user_classes` directory. This custom component is at the top level of the Flex application.

```
/server/flex>comp -o WEB-INF/flex/user_classes/MyOtherButton.swc -root .  
MyOtherButton.mxml
```

In this case, the root URL is the current directory, or `/flex`, and the output directory is in `user_classes`. Because the SWC file is output to the `user_classes` directory, the component author can immediately use the `MyOtherButton.swc` file in their Flex applications.

The author can also send the SWC file to other Flex application authors who can immediately use it after copying the SWC file to their `user_classes` directory. The new component requires no additional configuration. The only requirement is that a generic namespace be declared, as the following example shows:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" xmlns="*">  
  ...  
  <MyOtherButton/>  
  ...  
</mx:Application>
```

Using packaged components

Some components are created inside packages or directory structures so that they can be logically grouped and separated from application code. As a result, packaged components must have a namespace declaration that includes the package name or a unique namespace identifier that references their location within a package. In addition, if the SWC file compiles components from multiple packages, the component author can generate a manifest file when creating the SWC file which declares a generalized namespace for the components.

If you compile a SWC file from packaged component source files, component consumers must specify that package as part of the SWC file's namespace. The following example uses the `MyButton` component, which was created inside the `mycomponents` package. The component consumer specifies the package name in the namespace declaration:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"  
  xmlns:but="mycomponents.*">  
  ...  
  <but:MyButton/>
```

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

Using manifest files

Manifest files map a component namespace to class names. By default, the compc utility does not include a manifest file in the output SWC file. Some SWC files consist of multiple components from different packages, so you must generate a manifest file with your SWC file in some cases to prevent compiler errors.

The following example creates a new SWC file and manifest file for the MyButton1.mxml and MyButton2.mxml components:

```
/server/flex>compc -o ../WEB-INF/flex/user_classes/MyButton.swc
  -root .. -manifest mymanifest.xml myComponents/MyButton1.mxml
  moreComponents/MyButton2.mxml
```

This creates a SWC file in the target location as well as a manifest.xml file in the current directory. The manifest file defines the package names that the components used before being compiled into a SWC. The contents of the manifest file are as follows:

```
<?xml version="1.0"?>
<componentPackage>
  <component id="mycomponents.MyButton1" class="mycomponents.MyButton1"/>
  <component id="morecomponents.MyButton2" class="morecomponents.MyButton2"/>
</componentPackage>
```

Component consumers then add a new `<namespace>` element to the flex-config.xml file that points to the generated manifest file, as the following example shows:

```
<namespaces>
  <namespace uri="http://www.mycompany.com/mycomponent">
    <manifest>/WEB-INF/flex/mymanifest.xml</manifest>
  </namespace>
</namespaces>
```

The value of the `uri` attribute can be any nonzero-length string, but a good practice is to include a URL or other means of identifying the source of the component. It is also good practice to use a URI that resembles the ActionScript package name so that it can be easily identified. The URI must be unique within the current application. The URI does not have to point to an actual location but, rather, is meant to be human-readable.

The value of the `<manifest>` element is relative to the application root. In this case, the component user copied the component's manifest file to the *flex_app_root*/WEB-INF/flex directory.

When distributing the SWC file, the component author must include the manifest file with the SWC file. The component user must store the manifest file in the location specified by the `<manifest>` element. The component user also copies the SWC to the `user_classes` directory, or to any directory specified by the `<lib-path>` child tag in the flex-config.xml file.

In the application, the component user adds a namespace URI that matches the namespace defined in the manifest file; for example:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
  xmlns:but="http://www.mycompany.com/mycomponent">
  ...
  <but:MyButton/>
  ...
</mx:Application>
```

Distributing SWC files

After you generate a SWC file, you can use it in your Flex applications by copying it to the *flex_app_root*/WEB-INF/flex/user_classes directory. You can also copy SWC files to a directory specified by the `<lib-path>` child tag in the *flex-config.xml* file. SWC files must be stored at the top level of the *user_classes* directory or the directory specified by the `<lib-path>` element. You cannot store SWC files in subdirectories.

Note: Do not add SWC files to the *flex_app_root*/WEB-INF/flex/frameworks directory.

If the component does not have a namespace, you can add a generic namespace identifier in your `<mx:Application>` tag to use the component, as the following example shows:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
  xmlns:but="*">
```

If the component has a package name as part of its namespace, you must do one of the following:

- Add the package name to the namespace declaration in the `<mx:Application>` tag. For example:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
  xmlns:but="mycomponents.*">
```

- Create a manifest file with `compc` (using the `-manifest` option), add that file to the web application that uses the component, and refer to that manifest file in the *flex-config.xml* file. In the `<mx:Application>` tag, you specify only the unique namespace URI. For more information on specifying a namespace for the component, see [“Working with namespaces” on page 804](#).

For more information about distributing SWC files, see [“Distributing components” on page 840](#).

Editing the flex-config.xml file

Flex includes several configuration files to administer the behavior of your applications. Settings specific to the Flex application are defined in the *flex_app_root*/WEB-INF/flex/flex-config.xml file. Application server settings are located in the *flex_app_root*/WEB-INF/web.xml file.

The XML configuration files are easy to read and edit. In the case of the *flex-config.xml* file, you can use the *flex-config.xsd* schema file to validate your *flex-config.xml* settings. For *web.xml* file settings, consult your application server documentation.

You can change the location of the `flex-config.xml` file by editing the `web.xml` file and changing the value of the `flex.configuration.file` parameter. The following example shows the default settings:

```
<context-param>
  <param-name>flex.configuration.file</param-name>
  <param-value>/WEB-INF/flex/flex-config.xml</param-value>
  <description>configuration file</description>
</context-param>
```

You must restart the server after making changes to the `flex-config.xml` and `web.xml` files.

If you specify a URL beginning with a slash (/), it is relative to the web root. You can also use the `{context.root}` token in the URL to point to the context root of the application. This is defined by your application server. For more information on using the `{context.root}` token, see [“Editing the context root” on page 817](#).

Some tags in `flex-config.xml` use a `<path-element>` child tag to define values. These tags support relative paths and full URLs. The following example adds a URL to the SWC file in the `<lib-path>`:

```
<lib-path>
  <path-element>http://mydomain.com/foobar.swc</path-element>
</lib-path>
```

The following tags can take file system paths or URLs as values:

- `<lib-path>`
- `<debug-lib-path>`
- `<actionscript-classpath>`
- `<system-classes>`
- `<global-css-url>`

Setting production mode

Production mode is a state of the Flex application that you use when the application is running live on a public-facing server. It prevents the Flex application from generating profiling and debugging data.

The default value of `<production-mode>` is `false`. To enable production mode, change the value of the `<production-mode>` tag to `true`, as the following example shows:

```
<production-mode>true</production-mode>
```

When `<production-mode>` is `true`, all features in the `<debugging>` block are disabled. Flex also ignores query string parameter overrides such as `?debug=true` and `?asprofile=true`. When `<production-mode>` is `false`, individual features can be overridden on a per-request basis according to the settings in the `<debugging>` block.

Disabling production mode lets you override the following settings in the flex-config.xml file with query string parameters:

- asprofile
- debug
- showAllWarnings
- showBindingWarnings

In addition, when production mode is enabled, Flex only watches for changed files when the server starts up. Flex does not use the `<file-watcher-interval>` setting to continuously poll cached files in production mode.

Configuring caching

The Flex caching mechanism reduces compile-time delays by storing compiled SWF files in a content cache and responding to requests with the cached files when possible. In addition, Flex caches dependent files including components (SWC, MXML, and ActionScript files), CSS style sheets, images use in the `<mx:Image>` tag, and ActionScript files included with the `<mx:Script>` tag. If any of the dependent files changes, Flex recompiles the application.

The first time an MXML file is requested it will be compiled and cached. All subsequent requests return the cached item. The caching mechanism polls the MXML file and dependent files to determine if a new SWF file should be compiled.

SWF files are cached in memory and not on disk. Dependent files such as SWC files and ActionScript classes are cached on disk. Compiled custom components are cached as SWO files.

Caching is enabled by default. The default cache settings in flex-config.xml are as follows:

```
<cache>
  <cache-sws>true</cache-sws>
  <content-size>500</content-size>
  <mxml-size>500</mxml-size>
  <http-maximum-age>1</http-maximum-age>
  <file-watcher-interval>1</file-watcher-interval>
</cache>
```

The following table describes the child tags of `<cache>`:

| Child tag | Description |
|-----------------------------------|--|
| <code><cache-sws></code> | Set to <code>true</code> to enable caching of SWO files. A SWO file is a component file generated by Flex when you use custom MXML or ActionScript components. You can override this setting by appending <code>?sws=true false</code> to your query string. |
| <code><content-size></code> | The maximum number of cached SWF, SWD, and HTML files. The default value is 500. When the maximum number of files reaches the size limit, Flex removes the least-recently used files from the cache. |
| <code><mxml-size></code> | The maximum number of cached SWF files dynamically generated with the <code><mxml></code> tag in a JSP. The default value is 500. When the maximum number of files reaches the size limit, Flex removes the least-recently used files from the cache. |

| Child tag | Description |
|--|--|
| <code><http-maximum-age></code> | The maximum amount of time, in seconds, during which a cache may return its copy of content without checking for freshness of the document. The default value is 1 second. |
| <code><file-watcher-interval></code> | The amount of time Flex waits before polling files for changes to MXML and dependent files (such as MXML component files, SWC files, images, included *.as files, and ActionScript class files). When Flex finds that a file has changed, Flex recompiles the SWF file and replaces the cached SWF file with the new one. The default value is 1 second. |

You can force Flex to recompile the SWF file on a request for an MXML or SWF file by appending `?recompile=true` to your query string. To do this, the value of `process-debug-query-params` must be `true` and you cannot be in production mode.

You can also specify the number of embedded font faces to cache. For more information, see [“Editing font settings” on page 809](#).

Editing the global style sheet

You can specify the application’s global style sheet using the `<global-css-url>` tag in the `flex-config.xml` file. The style settings in the global style sheet are applied to all Flex components, but are overridden by local CSS settings.

The default value of `<global-css-url>` is as follows:

```
<global-css-url>/WEB-INF/flex/global.css</global-css-url>
```

The default `global.css` file is empty. You can specify a full path or a path that is relative to the application root or a full URL. In addition, you can use the `{context.root}` token in the value to reference the context root of the application.

For more information on using CSS in Flex applications, see [Chapter 16, “Using Styles and Fonts,” on page 395](#).

Editing font settings

You can specify caching, language and character ranges for embedded fonts in the `flex-config.xml` file using the `` tag and its child tags. The default fonts settings are as follows:

```
<font>
  <max-cached-fonts>20</max-cached-fonts>
  <max-glyphs-per-face>1000</max-glyphs-per-face>
  <language-range>
    <lang>en</lang>
    <range>U+0020-U+007E</range>
  </language-range>
</font>
```

The following table describes the child tags of the `` tag:

| Child tag | Description |
|--|---|
| <code><max-cached-fonts></code> | Sets the number of embedded font faces that Flex stores in its cache before rotating out older font faces. Each cached font uses up memory related to the size of the TrueType font (TTF) file. The default value is 20. When the number of embedded font faces exceeds <code>max-cached-fonts</code> , Flex removes the least-recently used from the cache. |
| <code><max-glyphs-per-face></code> | Defines the maximum number of character glyph outlines to cache for each font face. |
| <code><language-range></code> | <p>You can add one or more <code>language-range</code> blocks to your <code></code> tag. Each <code>language-range</code> block defines the language code and the range of characters available in that embedded font.</p> <p>The <code><lang></code> child tag specifies the reference name of the embedded font range.</p> <p>The <code><range></code> child tag defines the set of characters that are available in this embedded font's face.</p> <p>The following example creates an <code>englishRange</code> and an <code>otherRange</code> in the <code>flex-config.xml</code> file:</p> <pre> <language-range> <lang>englishRange</lang> <range>U+0020-U+007E</range> </language-range> <language-range> <lang>otherRange</lang> <range>U+00??</range> </language-range> </pre> |

Flex supports three default font faces, `_serif`, `_sans`, and `_typewriter`. For more information on using fonts in Flex, see [Chapter 16, “Using Styles and Fonts,” on page 395](#).

Editing debugging settings

To use debugging, you must set `<production-mode>` to `false`. If `<production-mode>` is set to `false`, you can override some of the `<debugging>` settings with query string parameters as long as `<process-debug-query-param>` is set to `true`. If `<production-mode>` is set to `true`, you cannot override any of the `<debugging>`.

Use the child tags of the `<debugging>` tag to configure debugging for all applications running on the Flex server. The `<debugging>` tag defines debug-related settings for the command-line debugger utility and compiler warnings.

The following table describes the debugging settings in `flex-config.xml`:

| Child tag | Description |
|--|--|
| <code><process-debug-query-param></code> | Set to <code>true</code> to allow query string parameters to override debugging settings. This setting does not supersede production mode. If production mode is enabled, this setting has no effect. The default value is <code>true</code> . If the tag is not specified, the default value is <code>false</code> . |
| <code><generate-debug-swfs></code> | Set to <code>true</code> to generate SWD files for use with the command-line debugger. You can override this setting by appending <code>?debug=true false</code> to the query string. The default value is <code>false</code> . For more information on using the command-line debugger, see Chapter 33, “Debugging Flex Applications,” on page 745 . |
| <code><generate-profile-swfs></code> | Set to <code>true</code> to generate SWF and SWD files for use with the Flex profiling tool. You can override this setting by appending <code>?asprofile=true false</code> to the query string. The default value is <code>false</code> . For more information on using the Flex profiling application, see Chapter 34, “Profiling ActionScript,” on page 769 . |
| <code><keep-generated-as></code> | Set to <code>true</code> to write <code>filename-generated.as</code> file to disk when Flex compiles the SWF file from an MXML application. The default value is <code>false</code> . |
| <code><keep-generated-swfs></code> | Set to <code>true</code> to write the generated SWF and SWD files to disk when Flex compiles the application. The default value is <code>false</code> . |
| <code><show-all-warnings></code> | Set to <code>true</code> to show all compiler warnings. You can override this setting by appending <code>?showAllWarnings=true false</code> to the query string. The default value is <code>true</code> . |
| <code><show-binding-warnings></code> | When <code><show-all-warnings></code> is <code>true</code> , this value controls whether or not binding warnings are shown. When <code><show-all-warnings></code> is <code>false</code> , this value has no effect. You can override this setting by appending <code>?showBindingWarnings=true false</code> to the query string. The default value is <code>true</code> . |
| <code><show-override-warnings></code> | When <code><show-all-warnings></code> is <code>true</code> , this value controls whether compiler override warnings are shown. When <code><show-all-warnings></code> is <code>false</code> , this value has no effect. The default value is <code>true</code> . |
| <code><web-service-proxy-debug></code> | Set to <code>true</code> to display the web service proxy request and response on the server side as well as debug information in client-side tracing. When set to <code>true</code> , Flex logs the messages according to the settings in the <code><logging></code> section. The default value is <code>false</code> . For more information about logging, see “Configuring logging” on page 818 . |

| Child tag | Description |
|---|--|
| <code><http-service-proxy-debug></code> | Set to <code>true</code> to display the HTTP proxy request and response on the server side. When set to <code>true</code> , Flex logs the messages according to the settings in the <code><logging></code> section. The default value is <code>false</code> . For more information about logging, see “Configuring logging” on page 818 . |
| <code><remote-objects-debug></code> | Set to <code>true</code> to display the remote object request and response on the server side as well as debug information in client side tracing. The default value is <code>false</code> . |
| <code><show-stacktraces-in-browser></code> | Set to <code>true</code> to display stack traces in the browser. The default value is <code>true</code> . Note: To display AMF stacktraces in the browser, you must set <code><show-stacktraces></code> to <code>true</code> in the gateway-config.xml file. |
| <code><show-source-in-compiler-errors></code> | Set to <code>true</code> to display source code context lines in the error pages. The default value is <code>true</code> . |
| <code><log-compiler-errors></code> | Set to <code>true</code> to log compiler errors to the Logger as errors. The default value is <code>true</code> . |
| <code><create-compile-report></code> | Set to <code>true</code> to generate a compiler report. This report contains the stack tree for all dependent symbols used in the application. Flex stores it in the same directory as the MXML file as <code>app_name-report.xml</code> . The default value is <code>false</code> . |

The Flex debugging settings also support a configurable password, which lets you perform remote debugging tasks in the Flash IDE. Flex encrypts this password during remote debugging sessions.

The following example sets the value of the `<debug-swf-password>` child tag:

```
<compiler>
...
  <debug-swf-password>oU812</debug-swf-password>
...
</compiler>
```

The default value of `<debug-swf-password>` is an empty string. Removing the `<debug-swf-password>` child tag or setting `<debug-swf-password>` to any number of spaces are equivalent to an empty string (`""`). The default value of the password is an empty string.

For more information on using the ActionScript debugger utility, see [Chapter 33, “Debugging Flex Applications,” on page 745](#).

Keeping generated files

Flex generates temporary files when compiling an MXML application into a SWF file. When you are developing Flex applications, examining this file can help in the debugging process. These include ActionScript class file, the SWF file itself, and, in some cases, the debug SWF/SWD file. By default, these files are not stored on disk, so they cannot be viewed.

The name of the temporary ActionScript class file is as follows:

mxml_filename-generated.as

For example, if your MXML file is named myApp.mxml, Flex generates a file called myApp-generated.as in addition to the application's SWF file.

You can enable or disable the storing of these files on disk with the `<keep-generated-as>` and `<keep-generated-swfs>` tags. Set the value to `true` to save the files, as the following example shows:

```
<keep-generated-as>true</keep-generated-as>
<keep-generated-swfs>true</keep-generated-swfs>
```

By default, the compiler writes these files into the same directory as the MXML file.

If you set `<keep-generated-swfs>` to `true`, Flex saves both the SWF and SWD files, if one was generated.

About deprecation

In some cases, Flex functionality has been deprecated. Deprecated features and properties have the following actions:

- Generate compilation warnings and display in the HTML wrapper for the application
- Continue to work in Flex 1.5
- Are removed from the product in a future major release

You can suppress deprecation warnings by setting `show-deprecated-warnings` to `false` in `flex-config.xml`.

Editing compiler settings

You change MXML compiler settings in the `flex-config.xml` file with the `<compiler>` child tags. You must restart your Java application server for changes to `flex-config.xml` to take effect.

Using the ActionScript optimizer

The ActionScript optimizer reduces the size of the Flex application's SWF file by 10–15%. It changes the generated ActionScript code prior to compilation, and can make debugging less reliable. As a result, you should disable optimization when you are debugging your MXML applications.

The optimizer removes trace statements entirely, including the side-effects of trace statements such as assignments. For example, suppose you have the following code:

```
x=1;
trace(x=2);
```

If optimization is enabled, `x` equals 1 because the trace statement is removed. If optimization is disabled, `x` equals 2 because the trace statement remains.

Use the `<optimize>` tag to enable or disable ActionScript optimization. The default value of the `<optimize>` tag is `true`. Set the `<optimize>` tag to `false` to disable optimization, as the following example shows:

```
<compiler>
...
<optimize>false</optimize>
...
</compiler>
```

Flex ignores the `optimize` setting if it generates a SWF file for debugging or profiling.

Using profiling

When profiling is enabled, Flex generates profiler data for Flex applications. Use the `<generate-profile-swfs>` tag to enable or disable ActionScript profiling for all applications running on the Flex server. The default value of the `<generate-profile-swfs>` tag is `false`.

To use profiling, you must set `<production-mode>` to `false`. If production mode is enabled, Flex does not generate profiling information and you cannot override it with query string parameters.

Set the `<generate-profile-swfs>` tag to `true` to generate data for the Flex ActionScript Profiler, as the following example shows:

```
<compiler>
...
<generate-profile-swfs>true</generate-profile-swfs>
...
</compiler>
```

You can override the MXML compiler setting for an individual application by appending `?asprofile=true|false` to the end of the query string, as the following example shows:

`http://localhost:8101/MyApp.mxml?asprofile=true`

In this case, Flex generates profiling data only for the specified application.

If you set `<process-debug-query-param>` to `false`, you cannot override the `<generate-profile-swfs>` setting.

For more information on using the Profiler, see [Chapter 34, “Profiling ActionScript,” on page 769](#).

Editing the ActionScript classpath

Flex searches directories in the ActionScript compiler's classpath for classes that are used in your Flex applications. You can edit the ActionScript classpath using the `<actionscript-classpath>` child tag of the `<compiler>` tag in `flex-config.xml`. They are searched in the order in which they appear in the `flex-config.xml` file.

Add a new `<path-element>` entry for each additional directory that you want to add to the ActionScript classpath. The value of `<path-element>` can be a full path or a path that is relative to the application root.

The default ActionScript compiler settings in `flex-config.xml` are as follows:

```
<compiler>
  <actionscript-classpath>
    <path-element>/WEB-INF/flex/frameworks</path-element>
    <path-element>/WEB-INF/flex/user_classes</path-element>
  </actionscript-classpath>
</compiler>
```

For more information on the ActionScript classpath, see *Getting Started with Flex*.

You can also add a child tag that points to a new location for the system classes. This can be useful if you have several individual Flex-enabled applications that point to a central location for common resources.

Note: The default system classpath is `flex_app_root/WEB-INF/flex/system_classes`. Do not add your custom classes to the directories specified in the `<system-classes>` child tag.

Changing the `<system-classes>` setting is for advanced users.

The following example adds a new `<system-classes>` setting:

```
<compiler>
  ...
  <system-classes>/flex-framework/system_classes</system-classes>
  ...
</compiler>
```

The `<compiler>` tag also defines `<lib-path>` elements, which are where you store custom components (or SWC files).

Setting the `<lib-path>`

Use the `<lib-path>` child tag of the `<compiler>` tag to specify path locations of component libraries, including SWC files, MXML components, and ActionScript components.

The default `<lib-path>` settings are as follows:

```
<lib-path>
  <path-element>/WEB-INF/flex/frameworks</path-element>
  <path-element>/WEB-INF/flex/user_classes</path-element>
</lib-path>
```

To add an additional directory to the `<lib-path>`, add a `<path-element>` tag that points to the new directory relative to the application root.

Note: Do not store custom SWC files in the `WEB-INF/flex/frameworks` directory. This is for internal use only.

During compilation, Flex merges all assets found in SWC files together and resolves which symbols in those files to use with a priority and version algorithm. The order of the SWC files in the `<lib-path>` tag or `user_classes` directory is ignored.

If two files are of the same priority, Flex chooses the one with the most recent timestamp.

Enabling accessibility

The Flex accessibility option extends support for accessible Flash components to Flex applications. Enabling accessibility in Flex provides the same functionality as adding the following ActionScript statements in a SWF file:

```
mx.accessibility.WindowAccImpl.enableAccessibility();
mx.accessibility.SimpleButtonAccImpl.enableAccessibility();
```

You can enable and disable the accessibility features of Flex components using the `<accessible>` tag in the `flex-config.xml` file. The default value of the `<accessible>` tag is `false`. This tag is a child tag of the `<compiler>` tag.

Set the `<accessible>` tag to `true` to enable accessibility features, as the following example shows:

```
<accessible>true</accessible>
```

You can override this setting by appending `?accessible=true/false` to the request's query string, as the following example shows:

```
http://mydomain.com/flex/myApp.mxml?accessibility=false
```

This override applies to the request, regardless of whether production mode is enabled.

For more information on using the accessibility classes, see *Flex ActionScript Language Reference*.

Configuring headless servers

A headless server is one that is running on UNIX or Linux and often does not have a monitor, keyboard, mouse, or even a graphics card. Headless servers are most commonly encountered in ISPs and ISVs, where available space is at a premium and servers are often mounted in racks. Enabling headless mode reduces the graphics requirements of the underlying system and can make for a more efficient use of memory.

To enable headless mode of the Flex application, define the value of the `<headless-server>` tag in the `flex-config.xml` file. Setting this property to `true` is required to support fonts and SVG images in a nongraphical environment. The `<headless-server>` tag is a child tag of `<compiler>`. The following example sets `<headless-server>` to `true`:

```
<headless-server>true</headless-server>
```

The default is commented out.

Setting the value of `<headless-server>` to `true` in `flex-config.xml` sets the system property `java.awt.headless` to `true`.

Setting the file encoding

You use the `<actionscript-file-encoding>` tag to set the file encoding in `flex-config.xml` so that the Flex compiler can correctly interpret ActionScript files. This tag does not affect MXML files because they are XML files that contain an encoding specification in the `<xml>` tag.

You use the `<actionscript-file-encoding>` tag when your ActionScript files do not contain a Byte Order Mark (BOM), and the files use an encoding that is different from the default encoding of your computer. If your ActionScript files contain a BOM, the compiler uses the information in the BOM to determine the file encoding.

For example, if your ActionScript files use Shift_JIS encoding, have no BOM, and your computer uses ISO-8859-1 as the default encoding, you use the `<actionscript-file-encoding>` tag as the following example shows:

```
<actionscript-file-encoding>Shift_JIS</actionscript-file-encoding>
```

Changing application server settings

This section describes how to make changes to your application server settings.

Editing the context root

The context root maps requests to the Flex application. In practice, the context root defines the URL prefix that clients use to request files in your application.

For example, the context root in the following URL is `/flex`:

```
http://localhost:8101/flex/myApp.mxml
```

The value is equivalent to a call to the `request.getContextPath()` method (Java) or the `HttpRequest.ApplicationPath` property (.Net).

If you are running Flex on the JRun application server, the context root for Flex is set in the *jrun_root/server_name*/WEB-INF/jrun-web.xml file. To change the context root, edit the `<context-root>` tag. The following example changes the context root to `/`:

```
<context-root>/</context-root>
```

The default context root is `/flex`. You can refer to the context root in the Flex configuration files using the `{context.root}` token. The value of `{context.root}` includes the prefix `"/"`. As a result, you are not required to add a forward slash before the `{context.root}` token.

If `{context.root}` is used in a nonrelative path, it must not have a leading `"/"` in front of it. For example, instead of this:

```
http://localhost/{context.root}
```

Do this:

```
http://localhost{context.root}
```

Nonrelative URLs are often used for the `<url>` and `<https-url>` elements in the data service sections of the `flex-config.xml` file.

To change the context root for non-JRun web application servers, consult your application server documentation.

In MXML tags, you can use `@ContextRoot()` to specify the context root in a URL, as the following example shows:

```
<mx:HTTPService url="@ContextRoot()/directory/myfile.xml"/>
```

Using virtual directories

Virtual directories map a request path to a real path in your file system. You can keep your files in directories outside of the application directory structure, but map them to a path inside the application directory structure.

For example, you can store files in an images folder at C:/images/production and use those images in your applications by adding a virtual directory that points to the /flex/images/production folder.

Note: In Windows systems, use forward slashes for path separators in XML configuration files.

To add a virtual directory on the JRun application server, add a virtual mapping in the *jrun_root/server_name/WEB-INF/jrun-web.xml* file. The following example adds the images virtual directory:

```
<virtual-mapping>
  <resource-path>/flex/images/*</resource-path>
  <system-path>c:/images/production</system-path>
</virtual-mapping>
```

To add virtual directories for non-JRun web application servers, consult your application server documentation.

Configuring logging

The Flex logging mechanism displays and records application-level events. These events include informational, error, warning, and debug events about the MXML applications running on the current server. They also optionally include low-level runtime compiler error messages.

The logging mechanism outputs to both the console window (System.out) and a log file defined in the flex-config.xml file. You can disable one or both of these, or change the level of logging. You can also define what levels of log entries to log, and define the file size and rotation frequency of the log files.

The default log file is located in *flex_app_root/WEB-INF/flex/logs/flex.log*.

About logging messages

Flex logs a timestamp, log level, and the log message for each entry in the log file or in the console. The Flex logging messages take the following form:

```
{date MM/dd HH:mm:ss} {log.level} {log.message}
```

For example:

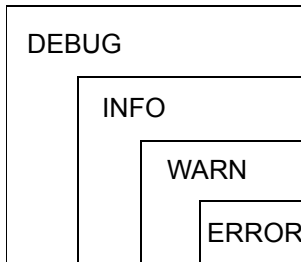
```
08/19 14:17:12 ERROR The detailed error message.
```

You cannot change the log file format.

About logging levels

There are four logging levels in Flex, from highest (or least restrictive) to lowest (or most restrictive): DEBUG, INFO, WARN, and ERROR. You select one level for each of the loggers that you use (console and file).

The following figure shows the logging level hierarchy:



If you select the lowest level, **ERROR**, Flex only logs **ERROR** messages. If you select the highest level, **DEBUG**, Flex logs all messages, including those for **DEBUG**, **INFO**, **WARN**, and **ERROR**.

In a production environment, Macromedia recommends that you set the logging level to the most restrictive level possible for performance reasons.

The following table describes the logging levels:

| Logging level | Description |
|---------------|--|
| DEBUG | DEBUG messages indicate internal Flex activities, such as bindable property chains. Select the DEBUG logging level to include DEBUG , INFO , WARN , and ERROR log messages in your log files. |
| INFO | INFO messages indicate general information to the developer or administrator, such as a message indicating that the Flex application has started. Select the INFO logging level to include INFO , WARN , and ERROR log messages in your log files. |
| WARN | WARN messages indicate that Flex encountered a problem with the application, but it will not stop running. Select the WARN logging level to include WARN and ERROR log messages in your log files. |
| ERROR | ERROR messages indicate when a critical service is not available or a situation has occurred that restricts use of the application. Select the ERROR logging level to include only ERROR log messages in your log files. |

Configuring logging

You configure the Flex logging settings in the `flex-config.xml` file. You can configure where Flex writes the log entries, plus determine whether Flex writes compiler errors. The `flex-config.xml` file is located in the `flex_app_root/WEB-INF/flex` directory. The settings apply to all Flex applications found under the `flex_app_root` directory.

The following example shows the default logging settings in the `flex-config.xml` file:

```
<logging>
  <level>info</level>
  <console>
    <enable>true</enable>
  </console>
  <file>
```

```

    <enable>true</enable>
    <file-name>/WEB-INF/flex/logs/flex.log</file-name>
    <maximum-size>200KB</maximum-size>
    <maximum-backups>3</maximum-backups>
  </file>
</logging>

```

The value of `<file-name>` must be an absolute path or it must start with a forward slash (/). If the location is invalid, Flex logs an error to the console and the application continues without file logging. On UNIX, if you start the path with a forward slash and one of the parent directories (other than root '/') exists, then the path is assumed to be absolute. Otherwise, it is assumed to be relative to the application.

To log compiler errors to the log file or console, you must set `<production-mode>` to `false`. If production mode is enabled, Flex does not generate log entries for compiler errors. Set the `<log-compiler-errors>` setting in the `<debugging>` section to `true` to log compiler errors to the logger.

The `<level>` child tag defines the level of detail that Flex logs. It takes one of the following values:

- INFO
- DEBUG
- WARN
- ERROR

When you set the logging level, it applies to both console and file logging. For more information, see [“About logging levels” on page 818](#).

The following sections describe the settings for the `<console>` and `<file>` tags.

For more information on using the `flex-config.xml` file, see [“Editing the flex-config.xml file” on page 806](#).

Console settings

Flex writes log entries to the console in which it was launched. You use the `<console>` tag to enable or disable console logging. The following table describes the console logging tag in the `flex-config.xml` file:

| Tag | Description |
|-----------------------------|---|
| <code><enable></code> | Enables or disables the console logging. Set to <code>true</code> to enable. Set to <code>false</code> to disable. The default value is <code>true</code> . |

File settings

Flex writes log entries to a log file. You use the `<file>` tag and its child tags to define the file logging settings in the `flex-config.xml` file.

The following table describes the file logging tags in the `flex-config.xml` file:

| Tag | Description |
|--------------------------------------|---|
| <code><enable></code> | Enables or disables file logging. Set to <code>true</code> to enable. Set to <code>false</code> to disable. The default value is <code>true</code> . |
| <code><file-name></code> | Specifies the path to the Flex log file. You can set the filename to either an absolute path or a relative path. A relative path is relative to the Flex application root. The default is <code>/WEB-INF/flex/logs/flex.log</code> . If you set an absolute path, you must create the directories in which the file resides before Flex adds log entries to that file. |
| <code><maximum-size></code> | <p>Specifies the size a log file reaches before Flex rotates it. You can specify KB, MB, or GB for kilobytes, megabytes, and gigabytes.</p> <p>When the log file reaches this size, Flex renames it to <code>flex_log[n]</code>, where <i>n</i> is the number of the backup on disk. Flex then begins a new log file.</p> <p>If <code><maximum-backups></code> is set to 0, Flex does not rotate log files.</p> <p>The default log file size is 200 KB.</p> |
| <code><maximum-backups></code> | Specifies the number of backup log files Flex keeps on disk. The default value is 3. Flex stores one current log file and up to three backup log files. Set to 0 to disable log file backups. |

CHAPTER 37

Applying Flex Security

Macromedia Flex security is based on the Macromedia Flash Player security model, which runs the player within a security sandbox. All Flex applications are sent to the client as a SWF file, therefore, Flex applications benefit from the robust security sandbox model used by standard Flash applications.

In addition, the Flex application is a J2EE-compliant web application, and as such, it can use any security implementation that is appropriate for that environment.

Contents

| | |
|--|-----|
| Flex security features | 823 |
| Flash Player security features | 825 |
| Security concerns of an open format technology | 832 |
| Resources | 833 |

Flex security features

The Flex security model protects both client and the server. There are two general aspects to security to consider when deploying Flex applications:

- Authorization and authentication of users accessing a server's resources
- Flash Player operating in a sandbox on the client

Flex is a J2EE application, so it supports working with the web application security of any J2EE application server. You use J2EE authentication and authorization techniques to prevent users from accessing your applications. The Flex application includes several built-in security mechanisms that let you control access to web services, HTTP services, and Java classes such as EJBs.

The Flash Player runs inside a security sandbox that prevents the client from being hijacked by malicious application code. This sandbox prevents a user from running a Flex application on their machine that can access system files and perform other tasks.

Overview of J2EE security

To handle authentication and authorization of the system, J2EE security covers multiple areas, each of which is handled by different roles:

- **Role definition and programmatic security** The application developer defines the roles that apply at the web application or EJB level. The application developer can optionally implement programmatic security, as required by the application. Declarative security is recommended over programmatic security for most J2EE applications.
- **Client coding** The application developer ensures that web application and EJB clients pass the required credentials (typically a user ID and password) at the appropriate time. Web application clients prompt for user ID and password. EJB clients either use credentials from the calling web application component or pass user ID and password as properties to the `InitialContext` constructor.
- **Role resolution and declarative security** The application assembler resolves and links programmer-defined roles with system roles. The application assembler also specifies declarative security in the web and EJB deployment descriptors.
- **Security architecture and user-store management** The administrator controls the user store, coordinates global role definition, and customizes security to match the site-specific security environment.

For information on using J2EE security methods, see your application server's documentation.

Web services and HTTP services

Flash Player operates within a security sandbox that limits what Flex applications and other Flash applications can access over HTTP. Flash applications are only allowed HTTP access to resources on the same domain from which they were served. This presents a problem for using web services and the `<HTTPService>` tag, since they typically get data from remote locations. Flex provides a proxy servlet that relays requests to remote web services and HTTP requests, redirects the requests, and then returns the responses to the client.

To prevent the proxy servlet from being used to stage denial of service (DOS) attacks or allow unauthorized access to services, Flex uses a *whitelist*. The whitelist is a list of URLs that the administrator explicitly gives the proxy servlet access to. Any URLs that are not included in the whitelist are not allowed to pass through the proxy servlet.

The default settings in `flex-config.xml` do not allow access to any URLs or resources using the proxies. You must explicitly add entries to the whitelists.

For more information, see [Chapter 30, “Using Data Services,” on page 655](#).

You can also assign access restrictions to entire applications or to any resource, such as web services or HTTP services, using the `security-constraint` and `web-resource-collection` elements in the `web.xml` file. For more information on using web application authentication, see your application server's documentation.

When you use the `<WebService>` and `<HTTPService>` tags, credentials are sent in base64 encoding. This is similar to how Basic authentication works. When connecting to secured data services, Macromedia recommends that you use https so that credential information is not visible. This is also the case with the data in an AMF packet for `RemoteObject`.

Java classes

You can control access to Java classes by using access settings, Basic authentication, and whitelists. Flex applications can use the `<mx:RemoteObject>` tag to call methods on Java objects that reside on the Java application server on which Flex is running. You should secure these objects from foreign access by using authentication.

For authentication of Java objects, Flex uses Basic authentication by default, in which the web browser displays a login dialog box. As an alternative, you can create a custom login dialog box in MXML to match the appearance of your application.

You can also assign access restrictions to entire applications or to any resource, such as Java classes, using the `security-constraint` and `web-resource-collection` elements in the `web.xml` file. For more information on using web application authentication, see your application server's documentation.

Flash Player security features

Flash Player has an extensive list of features that ensure Flash content is secure and safe. These include the following:

- Use the encryption capabilities of SSL in the browser to encrypt all communications between a Flash application and the server.
- An extensive sandbox security system that limits transfer of information that might pose a risk to security or privacy.
- Flash Player does not allow web content to read data from the local drive except for `SharedObjects` that were created by that domain.
- Flash Player cannot write any data to the disk except for data that is encapsulated in `SharedObjects`.
- Flash Player does not allow web content to read any data from a server that is not from the same domain, unless that content explicitly allows access.
- Flash Player does not allow web content to place more than 100K of data on the local disk from a single domain. The amount of data is configurable.
- Flash Player enables the user to disable the storage of information for any domain.
- Flash Player does not allow data to be sent from a camera or microphone unless the user gives permission.

About the security sandbox

The security sandbox defines a limited space in which a Flash application running within Flash Player is allowed to operate. Its primary purpose is to ensure the integrity and security of the client's machine, and as well as security of any Flash applications running in Flash Player.

The concept of the sandbox is simple. A Flash application executes inside a sandbox. Any information inside the sandbox can be communicated only to the domain from which the application was served. Access to information within the sandbox from outside of the sandbox is severely limited.

A Flash application's sandbox consists of the following:

- Everything contained in the SWF file
- User actions directed at the Flash application
- Servers in the domain from which the Flash application originated
- Local SharedObjects written by Flash applications from the same domain
- Limited configuration information about the computer on which the Flash application is running

In order to support local development and testing of Flash applications by developers, applications accessed as local files (either on the user's local disk or on LAN servers) have no sandbox limitations. This is consistent with the added caution users should always take when running any type of file locally.

About domain-based authentication

SWF files served from different domains cannot access each other's objects and variables. Nor can a SWF file served from one domain load data (using `loadVariables()`, for example) from another domain.

The reason for the domain limitation is to protect servers behind firewalls from being attacked by computers outside the firewall. A Flash application from `outside.hacker.org` must not be able to read data from any files stored on `inside.macromedia.com` when the application is running on a computer behind the Macromedia firewall.

Flash Player sandbox security model applies when loading a SWF file into an existing Flash application. However, applications loaded from separate domains exist within their own sandboxes. This isolates them from the other applications currently playing in the player. Content within an application's sandbox cannot penetrate outside the sandbox, and content outside the sandbox cannot access content within the sandbox.

All operations require an exact domain match. Similar domains, such as `www.mysite.com` and `store.mysite.com`, are not considered a match, nor are domains that are accessed with different port numbers. Identical numeric IP addresses are compatible. However, a domain name is not compatible with an IP address, even if the domain name resolves to the same IP address.

Content that is loaded through nonsecure (non-HTTPS) protocols cannot access content loaded through a secure (HTTPS) protocol, even when both are in the same domain, unless the server is configured to allow access. For example, a SWF file located at `http://www.macromedia.com/main.swf` cannot load data from `https://www.macromedia.com/data.txt`. This HTTPS restriction is asymmetrical; SWF files served over HTTPS can access other documents served over insecure protocols.

It is also possible for applications to communicate with each other through the LocalConnection object. The sandbox security model also applies to cross-application communication using this method. The key rule to remember is that an application in domain A cannot extract any information from an application in domain B, unless the application from domain B has given explicit permission for domain A to have access to it.

As mentioned previously, this rule must be strictly kept to prevent a SWF file on the public Internet from loading a SWF file from behind a firewall and extracting data.

Local file I/O access

Flash Player allows limited local file storage through the use of SharedObjects. SharedObjects, which you can think of in terms of web browser cookies, let developers store and retrieve information to the user's local file system.

SharedObjects have the following restrictions:

- Data is written to an unpredictable directory. The developer cannot control the location of that directory.
- The user has control of how much data can be stored, including the ability to disable storage on a per-domain basis.
- Data stored on the local file system is binary, serialized data controlled by Flash Player. The files all have a standard header and a *.so filename extension that prevents insertion of executable code or application data that might be inadvertently launched by the user.
- Data access is restricted by the domain-based authentication rules of Flash Player.

All file exchange through Flash Player is limited to an unpredictable directory on the client's machine that is created by Flash Player. The following limitations apply to the directory when being accessed from a Flash application playing in a web browser:

- The user controls how much information can be stored for a given domain.
- The default amount of data that each domain can store on the user's machine is 100K. Each SharedObject is stored in its own file. Each file is always counted as using at least 1000 bytes of data so that the default 100K limit allows up to 100 different SharedObjects for a single domain.

These restrictions ensure that the following conditions exist:

- Non-Flash information on the user's machine cannot be overwritten by a Flash application running in a browser.
- The threat of Denial of Service attacks, caused by filling the user's hard drive with data, is minimized.
- Foreign applications cannot easily find the location of the SharedObject data.
- Flash Player controls the format of the information stored on the file system by Flash Player. Developers do not have the ability to control the format of the data. The information is a binary serialization of the data being stored, and thus cannot be used for malicious attacks on the client machine.

- Data contained within SharedObjects can only be accessed under the domain-based rules discussed previously. For example, this means that data in a SharedObject set by an application from `www.domain1.com` cannot be accessed by an application from `www.domain2.com`.

Accessing external resources

Even with the restrictions, SWF files running within Flash Player can access external resources through the proxies defined in the `flex-config.xml` file and through other techniques such as tunneling and policy files. This section describes these techniques.

Using sandbox tunnelling

Code within a sandbox can only access global objects for that sandbox. However, it is sometimes necessary for two applications to reside in separate domains, yet access each other's data. For example, the Macromedia Answers Panel is loaded from the local disk by the Flash authoring tool, but might update itself by accessing the Macromedia website. In this case, it is necessary to allow an application loaded from `www.macromedia.com` to exchange data with the application loaded from the local disk.

Flex meets this need using the “tunneling” feature of the sandbox. Tunneling is implemented by the ActionScript method `System.security.allowDomain()` and has the following syntax:

```
System.security.allowDomain(domain1, ..., domainN);
```

This command grants `domain1` through `domainN` access to the sandbox of the SWF file that executes the command.

For example, the Answers Panel in the Flash MX authoring tool has a SWF file that is loaded as a local file. The Answers Panel SWF file that is loaded from `macromedia.com` needs to have access to the SWF file's variables. So the SWF file calls:

```
System.security.allowDomain("macromedia.com");
```

This command adds `macromedia.com` to the SWF file's “friends” list. Any SWF file loaded from `macromedia.com` or subdomains, such as `sub.macromedia.com`, can now access variables in the SWF file.

Access cannot be revoked once it has been granted and there is no way to retrieve a list of allowed domains.

The `allowDomain()` method permits *any* SWF file in the allowed domain to share data with *any other* SWF file in the domain permitting the access.

Using policy files

To make data and assets in runtime shared libraries available to SWF files in different domains, use a *cross-domain policy file*. A cross-domain policy file is an XML file that provides a way for the server to indicate that its data and documents are available to SWF files served from certain domains, or from all domains. Any SWF file that is served from a domain specified by the server's policy file is permitted to access data or assets from that server.

A Flash document can load data from an external source by using one of the following ActionScript methods:

- `loadVariables()`, `loadVariablesNum()`, `MovieClip.loadVariables()`, `LoadVars.load()`, `LoadVars.sendAndLoad()`
- `XML.load()`, `XML.sendAndLoad()`
- `XMLSocket.connect()`
- Symbol importing from runtime shared libraries
- Flash Remoting (`NetServices.createGatewayConnection()`)

These operations fail under the following conditions:

- When they reference a URL that is outside the exact domain of the SWF file that makes a request.
- When they reference an HTTPS URL where the SWF file that makes the request is not served over HTTPS.

When a Flash document attempts to access data from another domain, Flash Player attempts to load a policy file from that domain. If the domain of the Flash document that is attempting to access the data is included in the policy file, the data is automatically accessible.

Policy files function only on servers that communicate over HTTP, HTTPS, or FTP. The policy file is specific to the port and protocol of the server where it resides. For example, a policy file located at `https://www.macromedia.com:8080/crossdomain.xml` applies only to data loading calls made to `www.macromedia.com` over HTTPS at port 8080.

An exception to this rule is the use of an `XMLSocket` object to connect to a socket server in another domain. In that case, an HTTP server running on port 80 in the same domain as the socket server must provide the policy file for the method call.

The Flash Player ignores any policy file that is served using a cross-domain redirect. For example, if Flash Player request for `http://www.mysite.com/crossdomain.xml` redirects to `http://elsewhere.mysite.com/crossdomain.xml`, Flash Player ignores that policy file. Ensure that you do not use cross-domain redirects to serve policy files. (You can use redirects within the same domain.)

The default policy file is named `crossdomain.xml` and resides at the root directory of the server that is serving the data. You can use the `loadPolicyFile()` method to access a nondefault policy file.

In J2EE, web applications can have a different context roots, and you are not required to deploy any application to the default context root ("/"). This means that you cannot use a `crossdomain.xml` file in the web root without adding at least one web application at the default context root.

Using loadPolicyFile()

You can inform Flash Player of the location of a policy file using the `loadPolicyFile()` method. This method has the following syntax:

```
System.security.loadPolicyFile("URL")
```

The following example loads the policy file `pf.xml` from a subdirectory on the `foo.com` domain:

```
System.security.loadPolicyFile("http://foo.com/sub/dir/pf.xml")
```

Permissions granted by the policy file at that location apply to all content at the same level or below in the virtual directory hierarchy of the server.

You can load any number of policy files using the `loadPolicyFile()` method. When considering a request that requires a policy file, Flash Player waits for the completion of any policy file downloads before denying a request. If no policy file specified with `loadPolicyFile()` authorizes a request, Flash Player consults the default location, `/crossdomain.xml`.

Policy file syntax

The policy file is an XML file that contains a single `<cross-domain-policy>` tag, which in turn contains zero or more `<allow-access-from>` tags. A policy file that contains no `<allow-access-from>` tags has the same effect as not having a policy file on a server. The syntax is as follows:

```
<cross-domain-policy>
  <allow-access-from domain="IP_or_domain">
    [...]
</cross-domain-policy>
```

Each `<allow-access-from>` tag defines one property, `domain`, which specifies either an exact IP address, an exact domain, or a wildcard domain (any domain). Wildcard domains are indicated by either a single asterisk (*), which matches all domains and IP addresses, or an asterisk followed by a suffix, which matches only those domains that end with the specified suffix. Suffixes must begin with a dot. However, wildcard domains with suffixes can match domains that consist of only the suffix without the leading dot. For example, `foo.com` is considered to be part of `*.foo.com`. Wildcards are not allowed in IP domain specifications.

If you specify an IP address, access is granted only to SWF files loaded from that IP address using IP syntax (for example, `http://65.57.83.12/flashmovie.swf`), not those loaded using domain-name syntax. Flash Player does not perform DNS resolution.

Allowing access to HTTPS resources

By default, Flash applications served over HTTP cannot access HTTPS resources.

Each `<allow-access-from>` tag in the policy file can specify the `secure` property, which has two possible values, `true` and `false`. The default value is `true`. When you specify `secure="false"`, you allow access by Flash applications served over protocols other than HTTPS to access HTTPS resources.

Macromedia recommends against using `secure="false"` because allowing non-HTTPS documents to access HTTPS data can compromise HTTPS security. Instead, Macromedia recommends that you serve all SWF files that require access to HTTPS data over HTTPS. However, if using HTTPS for all of your documents is impractical, `secure="false"` will override the Flash Player default HTTPS protection.

Policy file example

The following example policy file permits access to Flash documents that originate from `foo.com`, `friendOfFoo.com`, `*.foo.com`, and `105.216.0.40`:

```
<?xml version="1.0"?>
<!-- http://www.foo.com/crossdomain.xml -->
<cross-domain-policy>
  <allow-access-from domain="www.friendOfFoo.com" />
  <allow-access-from domain="*.foo.com" />
  <allow-access-from domain="105.216.0.40" />
</cross-domain-policy>
```

Using web services and HTTP services

Flex applications support web service requests and results that are formatted as Simple Object Access Protocol (SOAP) messages and are transported over HTTP. HTTP services provide a way to include data accessed over HTTP in a Flex application.

Both HTTP services and web services require special consideration so that they can execute requests outside the Flash sandbox of the machine that served the application. Flex provides a proxy servlet that intercepts requests to remote web services and HTTP services, redirects the requests, and then returns the responses to the client.

To prevent the proxy servlet from being used to stage Denial Of Service (DOS) attacks or allow unauthorized access to services, Flex uses *whitelists*. The whitelist is a list of URLs that the administrator explicitly gives the proxy servlet access to. URLs that are not included in the whitelists are not allowed to pass through the proxy servlet. The default settings in `flex-config.xml` do not allow access to any URLs or resources using the proxies. For both named and unnamed services, only the URLs that the Flex administrator explicitly adds can pass through the proxy servlet.

You can obscure IP addresses used for HTTP services and web services by defining the services in your `flex-config.xml` file and using the service names in your MXML code. However, Flex passes the service name to the proxy servlet, which in turn looks up the WSDL URL in the service entry. The result is that the WSDL sent to the client still contains the actual endpoint URI in plain text. So while using named services is a good practice for obfuscating endpoints, it is not a complete solution.

You configure the web services and HTTP services whitelists in the `flex-config.xml` file. For more information on configuring web services and HTTP Services in Flex, see [Chapter 30, “Using Data Services,” on page 655](#).

Compatibility with older Players

In Flash Player 7 and later, if a version 6 (or earlier) SWF file attempts to load data from a server that resides in another domain, and that server doesn't provide a policy file that allows access from that SWF file's domain, then the Flash Player Settings dialog box appears. The dialog box asks the user to allow or deny the cross-domain data access.

If the user clicks the Allow button, the SWF file is permitted to access the requested data; if the user clicks Deny, the SWF file is not allowed to access the requested data.

To prevent this dialog box from appearing, create a security policy file on the server providing the data.

Security of data transport

A SWF file playing in a browser has many of the same security concerns as an HTML page being displayed in a browser. This includes the security of the SWF file while it is being loaded into the browser, as well as the security of communication between Flash and the server after the SWF file has loaded and is playing in the browser. In particular, data communication between the browser and the server is susceptible to being intercepted by third parties. The solution to this issue in HTML is to encrypt the communication between the client and server in order to make any data captured by third parties undecipherable and thus unusable. This is done by using an SSL-enabled browser and server.

Because a SWF file running within a browser uses the browser for almost all of its communication with the server, it can take advantage of the browser's built-in SSL support. This lets communication between the SWF file and the server be encrypted. Furthermore, the actual bytes of the SWF file are encrypted while they are being loaded into the browser. Thus, by playing a SWF file within an SSL-enabled browser through an HTTPS connection with the server, you can ensure that the communication between Flash Player and the server is encrypted and secure.

The one exception to this is the way Flash uses persistent sockets (through the ActionScript XMLSocket object), which does not use the browser to communicate with the server. Because of this, SWF files cannot take advantage of the built-in encryption capabilities of the browser. However, it is possible to use one-way encryption algorithms written in ActionScript to encrypt the data being communicated.

MD5 is a one-way encryption algorithm described in RFC 1321. This algorithm has been ported to ActionScript, which enables developers to secure one-way data using the MD5 algorithm before it is sent from the SWF file to the server. For more information about RFC 1321, see www.faqs.org/rfcs/rfc1321.html or www.rsasecurity.com/rsalabs/faq/3-6-6.html.

Security concerns of an open format technology

Flash applications share many of the same concerns and issues as web pages when it comes to protecting the security of data. Because the SWF file format is an open format, it is possible to extract data and algorithms contained within a SWF file. This is similar to how HTML and JavaScript code can be easily viewed by users. However, SWF files make viewing the code more difficult. A SWF file is compiled and is not human-readable like HTML or JavaScript.

But security is not obtained through obscurity. A number of third-party tools can extract data from compiled SWF files. As a result, do not consider that any data, variables, or ActionScript code compiled into a Flash application are secure. There are, however, a number of techniques you can use to secure sensitive information and still make it available for use in your SWF files.

To help ensure a secure environment, use the following general guidelines:

- Do not hard-code sensitive information, such as user names, passwords, or SQL statements in SWF files.
- If your SWF file needs access to sensitive information, load the information into the SWF file from the server at runtime. The data will not be part of the compiled SWF file and thus cannot be extracted by decompiling the SWF file. Ensure that you use a secure transfer mechanism, such as SSL, when loading the data.
- Implement sensitive algorithms on the server instead of in ActionScript.
- Use SSL whenever possible.
- Only deploy your web applications from a trusted server. Otherwise, the server-side aspect of your application could be compromised.

Resources

The following table lists resources that you can use when developing secure applications in Flex:

| Name | Description | Link |
|---|--|--|
| Macromedia Security Topic Center | Contains links to other security resources. | www.macromedia.com/devnet/security/ |
| Macromedia Security Zone | Contains security bulletins and technical briefs about security issues. | www.macromedia.com/devnet/security/security_zone/ |
| Macromedia Flex | Contains information and resources for Flex. | www.macromedia.com/software/flex/ |
| Macromedia Flash Player | Contains information and resources for Flash Player. | www.macromedia.com/software/flashplayer/ |
| Macromedia Flash MX | Contains information and resources for the Flash MX authoring environment. | www.macromedia.com/software/flash/ |
| Macromedia Designer & Developer Center | Contains articles, tutorials, and other information on Macromedia product development. | www.macromedia.com/desdev/ |
| Macromedia Flash Player security e-mail address | This e-mail address can be used for questions or comments regarding Flash Player security. | flashplayer_security@macromedia.com |

CHAPTER 38

Deploying Applications

This chapter describes how to deploy Macromedia Flex applications and write custom HTML wrappers that display the application in a browser. This chapter also describes how to configure your environment to support Macromedia Flash Player detection, deployment, and auto-update.

Contents

| | |
|--|-----|
| About deploying | 835 |
| Adding Flex to your application server | 836 |
| Distributing components | 840 |
| Working with Flex applications | 842 |
| About the HTML wrapper | 845 |
| Flash Player detection and deployment | 853 |
| Managing Flash Player auto-update | 857 |
| Passing request data to Flex applications. | 859 |

About deploying

J2EE encourages encapsulation of a single web application through its WAR, JAR, and EAR file packaging. When you first install Flex, you should deploy it as an exploded archive to your application sever. The J2EE Flex application settings are stored in flex/WEB-INF/web.xml and flex-config.xml. The web.xml file uses Flex-specific naming conventions to avoid collision with other applications running on the same server.

There are several considerations when deciding how to deploy a Flex application. When you prepare Flex for deployment, consider the following:

- **Target server** You can add Flex to an existing web application or build the web application on top of Flex. For more information, see [“Adding Flex to your application server” on page 836](#).
- **Components** If you create components for others to use in their Flex applications, you can distribute them as SWC files, ActionScript files, or MXML files. For more information, see [“Distributing components” on page 840](#).

- **File types** Your application can consist of pregenerated SWF files, MXML files, JSP pages, packaged ActionScript class files, or SWC files, or a combination of these plus media files, external resources, and data files. For more information, see [“Working with Flex applications” on page 842](#).

Adding Flex to your application server

The way you add Flex to your application server often affects how you will deploy your applications to other servers. There are two primary approaches to adding Flex to an application server:

- **Create a new Flex web application** For clean installations that have no pre-existing web applications, you explode the flex.war file to your application server. You then add application files inside the Flex open directory structure.
- **Add Flex to an existing web application** Merge the Flex configuration settings with your existing web application. When you prepare to deploy your application on another server, you must include the expanded contents of the flex.war file underneath your application root.

In some cases, you will have developed applications with a Developer or Trial Edition of Flex, and want to load-test the application or deploy it on a customer-facing server. Because the Developer Edition of Flex limits the number of concurrent requests and the Trial Edition times out, you may be required to update your Flex license before going live. Flex includes a command-line utility for doing this.

The following sections describe Flex installation and deployment techniques and the syntax for using the Flex command-line license tool.

Creating a new Flex application

To create a new Flex application, deploy the flex.war file to your application server as an exploded archive. This WAR file contains the minimum amount of files necessary to create a new Flex application. You can then add MXML files to the /flex directory and request those applications in that context.

Explode the flex.war file to a new directory called /flex under your server root. You can change the default name of the Flex root directory using the Flex installation instructions, and edit the default mappings.

To start developing a new application in Flex, you add MXML, ActionScript, or other files to the /flex directory under the server root:

```
server_root/flex/myApp.mxml
```

For example, you add your files to the following location on a JRun server:

```
/jrun4/servers/flexserver/flex/myApp.mxml
```

You can create a subdirectory under /flex for each application, and use your server's request mappings to hide this structure from end-users.

To administer the Flex framework, you edit the `flex-config.xml` file, which is in the following location:

```
server_root/flex/WEB-INF/flex/flex-config.xml
```

Adding Flex to an existing web application

If you have a pre-existing web application and want to add Flex resources to it, copy the *flex_app_root/WEB-INF/flex* directory to your application's WEB-INF directory. In addition, you must merge your application's configuration files with the Flex-specific settings in the `web.xml` file from the `flex.war` file.

Your file structure should match the following:

```
/application_root/WEB-INF/web.xml  
  
/application_root/WEB-INF/flex/frameworks/mx.swc  
/application_root/WEB-INF/flex/frameworks_debug/mx.swc  
/application_root/WEB-INF/flex/jars/*  
/application_root/WEB-INF/flex/system_classes/*  
/application_root/WEB-INF/flex/user_classes/*  
/application_root/WEB-INF/flex/logs/*  
/application_root/WEB-INF/flex/global.css  
/application_root/WEB-INF/flex/flex-config.xml  
/application_root/WEB-INF/flex/mxml-manifest.mxml  
/application_root/WEB-INF/flex/gate-config.xml  
  
/application_root/WEB-INF/lib/commons-beanutils.jar  
/application_root/WEB-INF/lib/commons-collections.jar  
/application_root/WEB-INF/lib/flashgateway.jar  
/application_root/WEB-INF/lib/flex-bootstrap.jar
```

To run Flex applications with this structure, you add your MXML files to your application's root directory.

To deploy your application on another server, create a WAR file that contains your application files and the Flex files. You then deploy the new WAR file to the other server. The Flex files must be expanded into the Flex directory underneath your application's root, because a WAR file cannot contain another WAR file. When you are deploying the WAR file, you might be required to change the licensing information of the Flex WAR. For more information, see [“Changing Flex license keys” on page 840](#).

If you have multiple applications on the same server, you must add the Flex files to each of them to use Flex in that application.

To update the web.xml file:

1. Add the following context parameter:

```
<context-param>
  <param-name>flex.class.path</param-name>
  <param-value>./WEB-INF/flex/jars</param-value>
</context-param>
```

2. Add the following Flex filters:

```
<filter>
  <filter-name>FlexDetectionFilter</filter-name>
  <filter-class>flex.bootstrap.BootstrapFilter</filter-class>
  <init-param>
    <param-name>filter.class</param-name>
    <param-value>flex.server.j2ee.DetectionFilter</param-value>
  </init-param>
</filter>
<filter>
  <filter-name>FlexCacheFilter</filter-name>
  <filter-class>flex.bootstrap.BootstrapFilter</filter-class>
  <init-param>
    <param-name>filter.class</param-name>
    <param-value>flex.cache.CacheFilter</param-value>
  </init-param>
</filter>
```

3. Add the following filter mappings:

Note: The FlexDetectionFilter must appear before the FlexCacheFilter mappings. The order of the filter mappings is important. The FlexCacheFilter is mapped to two servlets.

```
<filter-mapping>
  <filter-name>FlexDetectionFilter</filter-name>
  <servlet-name>FlexMxmlServlet</servlet-name>
</filter-mapping>
<filter-mapping>
  <filter-name>FlexCacheFilter</filter-name>
  <servlet-name>FlexMxmlServlet</servlet-name>
</filter-mapping>
<filter-mapping>
  <filter-name>FlexCacheFilter</filter-name>
  <servlet-name>FlexSwfServlet</servlet-name>
</filter-mapping>
```

4. Add the following servlet definitions:

- FlexProxyServlet
- FlexErrorServlet
- FlexInternalServlet
- FlexSwfServlet
- FlexMxmlServlet
- AMFGatewayServlet

- FlexForbiddenServlet
- session

Copy the servlet definitions from the default Flex web.xml file.

5. Add the following servlet mappings:

Note: The FlexSwfServlet has two mappings.

```
<!-- Flash Remoting AMF Gateway URL -->
<servlet-mapping>
  <servlet-name>AMFGatewayServlet</servlet-name>
  <url-pattern>/amfgateway/*</url-pattern>
</servlet-mapping>
<!-- Flash Web Services Proxy URL -->
<servlet-mapping>
  <servlet-name>FlexProxyServlet</servlet-name>
  <url-pattern>/flashproxy/*</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>FlexErrorServlet</servlet-name>
  <url-pattern>/flex-error</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>FlexInternalServlet</servlet-name>
  <url-pattern>/flex-internal/*</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>FlexMxmlServlet</servlet-name>
  <url-pattern>*.mxml</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>FlexSwfServlet</servlet-name>
  <url-pattern>*.swf</url-pattern>
</servlet-mapping>
<!-- use the same servlet for retrieving SWD debugging files -->
<servlet-mapping>
  <servlet-name>FlexSwfServlet</servlet-name>
  <url-pattern>*.swd</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>FlexForbiddenServlet</servlet-name>
  <url-pattern>*.as</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>FlexForbiddenServlet</servlet-name>
  <url-pattern>*.swc</url-pattern>
</servlet-mapping>
```

6. (Optional) Add the following error page mappings:

```
<error-page>
  <error-code>403</error-code>
  <location>/flex-error</location>
</error-page>
<error-page>
  <error-code>500</error-code>
```

```

        <location>/flex-error</location>
    </error-page>
    <error-page>
        <exception-type>javax.servlet.jsp.JspException</exception-type>
        <location>/flex-error</location>
    </error-page>
    <error-page>
        <exception-type>javax.servlet.ServletException</exception-type>
        <location>/flex-error</location>
    </error-page>

```

7. Add the following tag library definition:

```

<taglib>
    <taglib-uri>FlexTagLib</taglib-uri>
    <taglib-location>/WEB-INF/lib/flex-bootstrap.jar</taglib-location>
</taglib>

```

Changing Flex license keys

Flex includes a command-line utility that you use to change your license key. You can use the `licensetool` utility to upgrade an existing Flex installation or update the Flex license key inside your application's WAR file.

Note: When preparing a WAR file for sale or deployment, consult your Macromedia licensing agreement to determine what your options are.

You access the `licensetool` command-line utility in the *flex_install_dir/bin* directory. The `licensetool` utility has the following syntax:

```
licensetool [-info] [-install licensekey] flex_app_root|web_application_root
```

Use the `install` option to change the license key in the given WAR file. The following example updates the Flex license key in a default JRun installation:

```
licensetool -install MYNEWKEY c:/jrun4/servers/default/flex
```

When Flex is deployed into a web application, you point to the application's root directory to change the license key. The following example updates the Flex license key when Flex is deployed as part of MyApplication:

```
licensetool -install MYNEWKEY c:/dev/MyApplication
```

The `info` option returns the current license information for the given WAR file.

Distributing components

Components encapsulate common functionality so that you can use them across applications. Components can take the form of UI controls created in Flash MX 2004, ActionScript classes, or MXML files.

Distributing SWC files

Flash MX 2004 exports components as component packages (SWC files). A SWC file contains all the code, SWF files, images, and metadata associated with the component so you can easily add it to your Flex environment. When you distribute a component as a SWC file, you only need to give your users the single file.

Note: Once you create a SWC file, you can rename the file but the tag name you use in your MXML file must match the Linkage Identifier in the original FLA file.

To use a SWC file in your Flex application, move it to a directory defined by the `<lib-path>` child tag in the `flex-config.xml` file. Macromedia recommends using the `flex_app_root/WEB-INF/flex/user_classes` directory to store all custom components and classes.

Even if the SWC file contains classes in packages, do not deploy the SWC file to a directory structure that matches the package name. Copy the SWC file to the top level of a directory defined by the `<lib-path>` settings.

The following shows the default `<lib-path>` settings in the `flex_app_root/WEB-INF/flex/flex-config.xml` file:

```
<lib-path>
  <path-element>/WEB-INF/flex/user_classes</path-element>
  <path-element>/WEB-INF/flex/frameworks</path-element>
</lib-path>
```

Note: Do not store custom components or classes in the `flex_app_root/WEB-INF/flex/frameworks` directory. This directory is for Macromedia classes and components.

All SWC files found in the `<lib-path>` are merged together and resolved using the priority and version number prior to compilation of the final SWF. The order of the `<path-elements>` is ignored.

You can include the contents of debug SWC files in your Flex applications by editing the `<path-elements>` of the `<debug-lib-path>` child tag. Debug SWC files are ignored unless the debug flag is set during compilation. When the debug flag is set, the MXML compiler merges these libraries with the regular libraries, but at a higher priority level.

The following is the default value of the `<debug-lib-path>` element:

```
<debug-lib-path>
  <path-element>/WEB-INF/flex/frameworks_debug</path-element>
</debug-lib-path>
```

Defining component namespaces

To use components in your Flex applications, you must specify a namespace. The namespace optionally defines a tag prefix and can point to a package's subdirectories or a wildcard indicating that the component is in the same directory as the MXML file.

You store most custom components in the `flex_app_root/WEB-INF/flex/user_classes` directory. In the case of custom Flash components, you can store the SWC file in another directory that you specify in the `<lib-path>` setting in the `flex-config.xml` file.

Components in the same directory as the application can use a namespace defined as either of the following:

- `xmlns="*"`
- `xmlns:prefix="*"`

The following example does not define a prefix for the component's tag:

```
<mx:Application xmlns:mx=http://www.macromedia.com/2003/mxml xmlns="*">
  <MyFirstComponent/>
</mx:Application>
```

The following example defines the prefix as `local` and uses that prefix in the component's tag:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
  xmlns:local="*">
  <local:MyFirstComponent />
</mx:Application>
```

ActionScript or MXML components can use a namespace that identifies the package in which the component resides. The following example defines a namespace as being in the `customComponents.objects` package:

```
<mx:Application xmlns:mx=http://www.macromedia.com/2003/mxml
  xmlns:custom="customComponents.objects.*">
  <custom:MyFirstComponent />
</mx:Application>
```

SWC files must reside at the top level of the `user_classes` or directory specified by the `<lib-path>` setting. As a result, their namespace declarations cannot include subdirectories.

Distributing MXML and ActionScript components

You can store MXML and ActionScript components in packages that define their class structure. When distributing these files, include the package directory structure.

To install MXML and ActionScript components, copy the file structure to *flex_app_root/WEB-INF/flex/user_classes* or another directory defined as part of the `<lib-path>`. The component can then be accessed by any Flex application.

Working with Flex applications

When a client requests an MXML file, the default behavior is for Flex to return an HTML wrapper that points to a generated SWF file. However, there are several other methods to return a Flex application to a requesting browser.

You can access Flex applications by requesting the SWF file directly, or you can request an HTML or other type of page that contains a reference to a pregenerated SWF file. You can use the default HTML wrapper or create a custom one. You can also request a JSP page that defines the MXML dynamically.

When deciding how to create your Flex application, and how you want your users to access Flex content, you should familiarize yourself with the following options:

- Passing in request data
- History management support
- Flash Detection and Deployment Kit support
- Customizing the appearance of your Flex application in the browser

The default HTML wrapper supports history management and the Flash Detection and Deployment Kit. To write a custom HTML wrapper, you must precompile your Flex application into a SWF file or reference the MXML files in a JSP page.

Precompiling SWF files

You can precompile SWF files from your MXML source code using the mxmcl compiler or requesting them in a browser and saving the generated files. After you compile the SWF file, you can create a custom HTML wrapper that returns the SWF file when the user requests that HTML file. This lets you add Flex applications to existing web pages. With a precompiled SWF, you can add the SWF file to any dynamic or static web page. You need only add the HTML wrapper code to that page.

Note: If you precompile a Flex SWF file, you must deploy it on a server running a licensed copy of Flex.

For more information on using mxmcl to precompile your SWF files, see [“Using the command-line compiler” on page 796](#).

To define a custom HTML wrapper for your precompiled SWF file, you can use the contents of the default HTML wrapper generated by Flex as a template. This wrapper includes support for history management, `flashVars` properties, and player detection.

To add the HTML wrapper to your application:

1. Request your MXML file in a browser. Flex returns the default HTML wrapper.
2. Select View > Source (in Internet Explorer) or View > Page Source (in Netscape).

The browser displays the HTML source code.

3. Copy the `<script>` and `<noscript>` blocks from the source code.

The code that you copy appears similar to the following:

```
<script language='javascript' charset='utf-8' src='/flex/flex-internal?action=js'></script>
<noscript>
  <object classid='clsid:D27CDB6E-AE6D-11cf-96B8-444553540000'
    codebase='http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version=7,0,14,0' width='500' height='500'>
      <param name='flashVars' value=''>
      <param name='src' value='MyApp.mxml.swf'>
      <embed pluginspage='http://www.macromedia.com/go/getflashplayer'
        width='500' height='500' flashVars='' src='file1.mxml.swf' />
    </object>
  </noscript>
```

```

<script language='javascript' charset='utf-8'>
    document.write("<object classid='clsid:D27CDB6E-AE6D-11cf-96B8-444553540000' codebase='http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version=7,0,14,0' width='500' height='500'");
    document.write(">");
    document.write("    <param name='flashVars' value='historyUrl=%2Fflex%2Fflex-internal%3Faction%3Dhtml&lconid=" + lc_id + "'>");
    document.write("    <param name='src' value='MyApp.mxml.swf'>");
    document.write("    <embed pluginspage='http://www.macromedia.com/go/getflashplayer' width='500' height='500'");
    document.write("        flashVars='historyUrl=%2Fflex%2Fflex-internal%3Faction%3Dhtml&lconid=" + lc_id + "'");
    document.write("        src='MyApp.mxml.swf'");
    document.write("    />");
    document.write("</object>");
</script>

<script language='javascript' charset='utf-8'>
    document.write("<br><iframe src='/flex/flex-internal?action=html' name='_history' frameborder='0' scrolling='no' width='22' height='0'></iframe><br>");
</script>

```

4. Paste this code into your HTML page.

For more information on defining a custom HTML wrapper, see [“About the HTML wrapper” on page 845](#). You can add additional logic for player detection and deployment to a custom wrapper as described in [“Adding detection and deployment to custom wrappers” on page 856](#).

Using JSPs

You can develop your Flex application using JSPs. This requires that you use the Flex JSP Tag Library. For more information on using the Flex JSP Tag Library, see [Chapter 35, “Using the Flex JSP Tag Library,” on page 781](#).

When you write MXML code in a JSP or reference external MXML files or SWF files, Flex adds the HTML wrapper to the output of the JSP.

There are several ways to create Flex applications with JSPs:

- Write MXML in the JSP. The JSP contains dynamic MXML written with the Flex JSP Tag Library; for example:

```

<%@ taglib uri="FlexTagLib" prefix="mm" %>
<mm:mxml>
    <mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml"
        width="200" height="240">
        <mx:Label id="label0" text="Hi <%= session.getAttribute("uname") %>"/>
    </mx:Application>
</mm:mxml>

```

- Use the `source` property of the `<mxml>` tag to include external MXML files in your JSPs; for example:

```
<%@ taglib uri="FlexTagLib" prefix="mm" %>
<mm:mxml source="../FlexApps/MyApp.mxml" />
```

- Use the `source` property of the `<flash>` tag to include a pregenerated SWF in your JSP; for example:

```
<%@ taglib uri="FlexTagLib" prefix="mm" %>
<mm:flash source="../FlexApps/MyApp.swf" />
```

The `<flash>` and `<mxml>` tags support a set of properties that define the presentation of the SWF file on the generated HTML page, such as background color and alignment. Setting a property adds that property to the `<object>` and `<embed>` tags within the HTML wrapper. In some cases, the properties are only used by the `<object>` or the `<embed>` tag, but not both.

The following example sets the background color of the pregenerated SWF file to `#FFCCCC`:

```
<mm:flash src="MyApp.swf" bgcolor="#FFCCCC" />
```

You can also set these properties with the `<param>` child tag of the `<flash>` and `<mxml>` tags.

The available display properties are listed in [“About the `<object>` and `<embed>` tag properties” on page 848](#).

To pass request data into a Flex application written as a JSP, you can add attributes in your JSP page with JSP expression syntax or use query string parameters. You can also take advantage of the Flex JSP Tag Library and use the `<flashvar>` child tag of the `<flash>` and `<mxml>` tags to pass user-defined data to the Flex application.

Note: Compilation performance depends on the number of unique MXML fragments in your source JSP. As a result, dynamic MXML should be used sparingly.

The value can be a JSP expression, as the following example shows:

```
<mm:flash src="MyApp.swf">
  <mm:flashvar name="firstname" value="<%= session.getAttribute('username')"/>
</mm:flash>
```

About the HTML wrapper

When you request an MXML file, Flex creates an HTML wrapper that references the generated SWF file. The HTML wrapper is not written to disk, but is stored in the cache and returned to the browser from the cache on each request. If the source files change, Flex regenerates the wrapper. Also, if the wrapper’s output includes content that is based on changing URL parameters, Flex regenerates the wrapper for each unique request.

The HTML wrapper consists of an `<object>` tag and an `<embed>` tag that format the SWF file on the page, define data object locations, and pass runtime variables to the SWF file. In addition, the HTML wrapper includes tags for history management. Flex also generates a second HTML wrapper that performs Flash Player version detection and deployment. This wrapper is invisible to end-users.

Flex also generates an HTML wrapper when you write MXML in your JSP pages using the Flex JSP tag library. The wrapper becomes part of the JSP servlet's output stream. Flex does not generate the HTML wrapper when you use the mxmmlc precompiler to create a SWF file from an MXML file.

Customizing the HTML wrapper

You can write your own HTML wrapper for your SWF files rather than use the ones generated by Flex. Your own wrapper can be any text file that contains the appropriate template variables. It can be a plain HTML file, a ColdFusion page, or an Active Server Page (ASP).

To write your own HTML wrapper, keep the following guidelines in mind:

- You must point to the SWF file. Set the value of the `src` property of the `<object>` tag to `mxml_filename.mxml.swf`.

The following example defines the `src` property of the `<object>` tag for an MXML application called `MyApp.mxml`:

```
<param name='src' value='MyApp.mxml.swf'>
```

The `<embed>` tag uses the `src` property to define the source of the SWF file:

```
src='MyApp.mxml.swf'
```

- If you use both the `<object>` and the `<embed>` tags in your custom wrapper, use identical values for each attribute to ensure consistent playback across browsers.
- To support history management, copy the script from the Flex-generated HTML wrapper and paste it into your custom wrapper.
- To support Flash Player detection and deployment, see the instructions in [“Adding detection and deployment to custom wrappers” on page 856](#).
- You cannot write a custom wrapper when using the `<flash>` and `<mxml>` tags to return the Flex application in a JSP. You can use these tags' `<param>` child tag to define some settings for the Flash application.
- The HTML wrapper uses the context root of the application in several places. If you change this from the generated wrapper, ensure that you change it in all locations.

About the `<object>` and `<embed>` tags

The `<object>` tag is used by Internet Explorer 3.0 or later on Windows 9x, Windows 2000, Windows NT, Windows ME, and Windows XP platforms or any browser that supports the use of the Flash ActiveX control. The `<embed>` tag is used by Netscape Navigator 2.0 or later, or browsers that support the use of the Netscape-compatible plug-in version of Flash Player.

To ensure that browsers display Flex applications properly, Flex nests the `<embed>` tag within the `<object>` tag. ActiveX-enabled browsers ignore the `<embed>` tag inside the `<object>` tag. Netscape and Microsoft browsers using the Flash Plug-in do not recognize the `<object>` tag, and read only the `<embed>` tag.

The following example shows the default HTML wrapper generated by Flex for an application called MyApp (without the history management-specific tags):

```
<object classid='clsid:D27CDB6E-AE6D-11cf-96B8-444553540000' codebase='http://
download.macromedia.com/pub/shockwave/cabs/flash/
swflash.cab#version=7,0,14,0' width='200' height='200'>
  <param name='flashVars' value=''>
  <param name='src' value='MyApp.mxml.swf'>
  <embed pluginspage='http://www.macromedia.com/go/getflashplayer'
    width='200' height='200' flashVars='' src='MyApp.mxml.swf' />
</object>
```

Four settings (height, width, classid, and codebase) are properties that appear within the `<object>` tag; all others are attributes that appear in separate, named `<param>` tags, as the following example shows:

```
<object classid="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000" width="100"
  height="100" codebase="http://active.macromedia.com/flash7/cabs/
  swflash.cab#version=7,0,14,0">
  <param name="src" value="movienamename.swf">
  <param name="play" value="true">
  <param name="loop" value="true">
  <param name="quality" value="high">
</object>
```

For the `<embed>` tag, all settings are properties that appear between the angle brackets of the opening `<embed>` tag, as the following example shows:

```
<embed src="movienamename.swf" width="100" height="100" play="true" loop="true"
  quality="high" pluginspage="http://www.macromedia.com/shockwave/
  download/index.cgi?P1_Prod_Version=ShockwaveFlash">
</embed>
```

To use both tags together, position the `<embed>` tag just before the closing `<object>` tag, as follows:

```
<object classid="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000" width="100"
  height="100" codebase="http://active.macromedia.com/flash7/cabs/
  swflash.cab#version=6,0,0,0">
  <param name="src" value="movienamename.swf">
  <param name="play" value="true">
  <param name="loop" value="true">
  <param name="quality" value="high">
  <embed src="movienamename.swf" width="100" height="100" play="true" loop="true"
    quality="high" pluginspage="http://www.macromedia.com/shockwave
    /download/index.cgi?P1_Prod_Version=ShockwaveFlash">
  </embed>
</object>
```

About the <object> and <embed> tag properties

When Flex generates an HTML wrapper, it includes a set of properties in the <object> and <embed> tags by default. If you write a custom HTML wrapper, or if you use the Flex JSP tag library, you can include additional properties that define the appearance of the SWF in the browser. This section describes the default properties, and then lists the additional display properties.

About the default properties

When Flex generates an HTML wrapper, it includes a subset of properties in the <object> and <embed> tags by default. These properties are the minimum set of properties required by Flash Player. You can set these properties either as <param> tags within the <flash> or <mxml> custom tags or as attributes of those tags. The following table describes the default properties included in the HTML wrapper:

| Property | Description |
|-----------------|---|
| classid | Defines the <code>classid</code> of Flash Player. This identifies the ActiveX control for the browser. Internet Explorer 3.0 or later on Windows 9x, Windows 2000, Windows NT, Windows ME, and Windows XP prompt the user with a dialog box asking if they would like to auto-install Flash Player if it's not already installed. This process can occur without the user having to restart the browser. This property is used for the <object> tag only. |
| codebase | Identifies the location of Flash Player ActiveX control so that the browser can automatically download it if it is not already installed. This property is used for the <object> tag only. |
| flashVars | Sends variables to the application. The format of the string is a set of name-value pairs, each separated by an ampersand (&). Browsers support string sizes of up to 64KB (65535 bytes) in length. The default value of this property is an empty string. For more information on using <code>flashVars</code> to pass variables to Flex applications, see “Using flashVars” on page 860 . |
| height width | Defines the height and width, in pixels, of the SWF file. Flex makes a best guess to determine the height and width of the application if none is provided. User agents scale an <code>object</code> or image to match the height and width specified by the author. Lengths expressed as percentages are based on the horizontal or vertical space currently available, not on the natural size of the <code>object</code> . You can also set the height and width of a Flex application by setting the <code>height</code> and <code>width</code> properties of the <flash> or <mxml> JSP tags or by setting the <code>height</code> and <code>width</code> properties of the <mx:Application> tag in an MXML file. |

| Property | Description |
|--------------------------|---|
| <code>pluginspage</code> | Identifies the location of Flash Player plug-in so that the user can download it if it is not already installed. This property is used for the <code><embed></code> tag only. |
| <code>src</code> | Identifies the location of the SWF file. If you write a custom wrapper but deploy your application as MXML files and not pregenerated SWF files, use the following naming convention: <i>movie_name.mxml.swf</i> This property is used for the <code><object></code> and <code><embed></code> tags. If you use this property with the <code><flash></code> JSP tag, it points to the SWF file. If you use it with an <code><mxml></code> JSP tag, it points to the MXML file. |

Additional display properties

In addition to the default properties generated by Flex for the HTML wrapper, you can add other properties that define the display of the SWF file in the browser. These properties are based on the HTTP specification for the `<object>` and `<embed>` tags. In all cases, you can add these to a custom HTML wrapper. In some cases, you can set these properties as attributes of the `<flash>` and `<mxml>` tags in the Flex JSP Tag Library or use `<param>` child tags to define them.

The following table describes optional display properties that you can add to your custom HTML wrapper:

| Property | Type | Description |
|--------------------------------|--------|--|
| <code>allowscriptaccess</code> | String | Controls the ability to perform outbound scripting from within a SWF file. The <code>allowscriptaccess</code> property can prevent a SWF file hosted from one domain from accessing a script in an HTML page that comes from another domain. Setting <code>allowscriptaccess</code> to <code>never</code> for all SWF files hosted from another domain can ensure security of scripts located in an HTML page. Possible values are as follows: <ul style="list-style-type: none"> <code>always</code>: Outbound scripting always succeeds. <code>never</code>: Outbound scripting always fails. The default is <code>always</code> . |
| <code>archive</code> | String | Specifies a space-separated list of URIs for archives containing resources used by the application, which may include the resources specified by the <code>classid</code> and <code>data</code> properties. Preloading archives can result in reduced load times for applications. Archives specified as relative URIs are interpreted relative to the <code>codebase</code> property. |
| <code>border</code> | int | Specifies the width of an <code>object</code> border, in pixels. The default value for this property depends on the user agent. |

| Property | Type | Description |
|-------------------------|---------|---|
| <code>codetype</code> | String | <p>Defines the content type of data expected when downloading the application specified by the <code>classid</code>.</p> <p>The <code>codetype</code> property is optional but recommended when the <code>classid</code> property is specified; it lets the browser avoid loading unsupported content types.</p> <p>The default value of the <code>codetype</code> property is the value of the <code>type</code> property.</p> |
| <code>data</code> | String | <p>Specifies the location of the application's data. For example, instance image data for objects defining images.</p> <p>If the <code>data</code> property is a relative URI, it is relative to the <code>codebase</code> property.</p> |
| <code>declare</code> | Boolean | <p>Makes the current <code>object</code> definition a declaration only. The <code>object</code> must be instantiated by a subsequent object definition referring to this declaration.</p> |
| <code>devicefont</code> | Boolean | <p>Specifies whether static text objects for which the Device Font option is not selected will be drawn using a device font anyway, if the fonts needed are available from the operating system.</p> |
| <code>dir</code> | String | <p>Specifies the base direction of text in an element's content and attribute values. It also specifies the directionality of tables.</p> <p>Possible values are <code>LTR</code> (left-to-right text or table) and <code>RTL</code> (right-to-left text or table).</p> |
| <code>hspace</code> | int | <p>Specifies the amount of white space inserted to the left and right of an object. The default value is not specified, but is generally a small, nonzero length.</p> |
| <code>id</code> | String | <p>Identifies the SWF file to the host environment (a web browser, for example) so that it can be referenced using a scripting language.</p> <p>The <code>id</code> property is only used with the <code><object></code> tag. It is equivalent to the <code>name</code> property used with the <code><embed></code> tag.</p> |
| <code>lang</code> | String | <p>Specifies the base language of an element's property values and text content.</p> <p>The default value is <code>unknown</code>. Language information specified using the <code>lang</code> property can be used by a user agent to control rendering in a variety of ways.</p> |
| <code>name</code> | String | <p>Identifies the SWF file to the host environment (a web browser, typically) so that it can be referenced using a scripting language.</p> <p>The <code>name</code> property is only used with the <code><embed></code> tag. It is equivalent to the <code>id</code> property used with the <code><object></code> tag.</p> |
| <code>standby</code> | String | <p>Defines a message that the user agent displays while loading the object's implementation and data.</p> |
| <code>style</code> | String | <p>Specifies style information for the current element.</p> <p>The syntax of the value of the <code>style</code> property is determined by the default style sheet language. In CSS, property declarations have the form "name:value" and are separated by a semicolon.</p> |

| Property | Type | Description |
|---------------------------|---------|--|
| <code>supportembed</code> | Boolean | Determines whether the Netscape-specific <code><embed></code> tag is supported. The <code>supportembed</code> property is optional, and the default value is <code>true</code> . Set to <code>false</code> to suppress the <code><embed></code> tag in the generated HTML. |
| <code>tabindex</code> | int | Specifies the position of the current element in the tabbing order for the current document. This value must be a number between 0 and 32767. User agents should ignore leading zeros. |
| <code>title</code> | String | Offers advisory information about the element for which it is set. Values of the <code>title</code> property can be rendered by user agents in a variety of ways. For example, visual browsers frequently display the title as a ToolTip. Audio user agents might speak the title information in a similar context. |
| <code>type</code> | String | Specifies the content type for the data specified by the <code>data</code> property. The <code>type</code> property is optional but recommended when data is specified; it prevents the browser from loading unsupported content types. If the value of this property differs from the HTTP Content-Type returned by the server, the HTTP Content-Type takes precedence. |
| <code>usemap</code> | String | Associates an image map with an element. The image map is defined by a <code>map</code> element. The value of <code>usemap</code> must match the value of the <code>name</code> attribute of the associated <code>map</code> element. |
| <code>vspace</code> | int | Specifies the amount of white space inserted above and below an object. The default value is not specified, but is generally a small, nonzero length. |

The following table describes the display properties that you can set using a `<param>` tag or as an attribute of the `<mxml>` and `<flash>` custom tags, in addition to manually adding them to your custom HTML wrapper:

| Property | Type | Description |
|--------------------|--------|--|
| <code>align</code> | String | Specifies the position of the object. The <code>align</code> property supports the following values: <ul style="list-style-type: none"> <code>bottom</code>: Vertically aligns the bottom of the <code>object</code> with the current baseline. This is the default value. <code>middle</code>: Vertically aligns the middle of the <code>object</code> with the current baseline. <code>top</code>: Vertically aligns the top of the <code>object</code> with the top of the current text line. <code>left</code>: Horizontally aligns the <code>object</code> to the left margin. <code>right</code>: Horizontally aligns the <code>object</code> to the right margin. |
| <code>base</code> | String | Specifies the base directory or URL used to resolve all relative path statements in the application. |

| Property | Type | Description |
|----------------------|---------|---|
| <code>bgcolor</code> | String | <p>Specifies the background color of the application. Use this property to override the background color setting specified in the Flash file. This property does not affect the background color of the HTML page.</p> <p>Valid formats for <code>bgcolor</code> are any <code>#RRGGBB</code>, hexadecimal, or RGB value.</p> |
| <code>menu</code> | Boolean | <p>Changes the appearance of the menu that appears when users right-click over a Flex application in Flash Player. Set to <code>true</code> to display the entire menu. Set to <code>false</code> to display only the About and Settings options on the menu.</p> <p>The default is <code>true</code>.</p> |
| <code>quality</code> | String | <p>Defines the quality of playback in Flash Player. Valid values of <code>quality</code> are <code>low</code>, <code>medium</code>, <code>high</code>, <code>autolow</code>, <code>autohigh</code>, and <code>best</code>. The default value is <code>best</code>.</p> <p>The <code>low</code> setting favors playback speed over appearance and never uses anti-aliasing.</p> <p>The <code>autolow</code> setting emphasizes speed at first but improves appearance whenever possible. Playback begins with anti-aliasing turned off. If Flash Player detects that the processor can handle it, anti-aliasing is turned on.</p> <p>The <code>autohigh</code> setting emphasizes playback speed and appearance equally at first, but sacrifices appearance for playback speed if necessary. Playback begins with anti-aliasing turned on. If the actual frame rate drops below the specified frame rate, anti-aliasing is turned off to improve playback speed. Use this setting to emulate the View > Antialias setting in Flash.</p> <p>The <code>medium</code> setting applies some anti-aliasing and does not smooth bitmaps.</p> <p>The <code>high</code> setting favors appearance over playback speed and always applies anti-aliasing.</p> <p>The <code>best</code> setting provides the best display quality and does not consider playback speed. All output is anti-aliased and all bitmaps are smoothed.</p> |
| <code>align</code> | String | <p>Positions the SWF file within the browser. Valid values are <code>L</code>, <code>T</code>, <code>R</code>, <code>B</code>, <code>TL</code>, <code>TR</code>, <code>BL</code>, and <code>BR</code>.</p> <p><code>L</code>, <code>R</code>, <code>T</code>, and <code>B</code> align the SWF file along the left, right, top or bottom edge, respectively, of the browser window and crop the remaining three sides as needed.</p> <p><code>TL</code> and <code>TR</code> align the SWF file to the top left and top right corner, respectively, of the browser window and crop the bottom and remaining right or left side as needed.</p> <p><code>BL</code> and <code>BR</code> align the SWF file to the bottom left and bottom right corner, respectively, of the browser window and crop the top and remaining right or left side as needed.</p> |

| Property | Type | Description |
|--------------------|--------|--|
| <code>scale</code> | String | <p>Defines how the browser fills the screen with the SWF file. The default value is <code>showall</code>. Valid values of the <code>scale</code> property are <code>showall</code>, <code>noborder</code>, and <code>exactfit</code>.</p> <p>Set to <code>showall</code> to make the entire SWF file visible in the specified area without distortion, while maintaining the original aspect ratio of the SWF. Borders may appear on two sides of the SWF file.</p> <p>Set to <code>noborder</code> to scale the SWF file to fill the specified area, without distortion but possibly with some cropping, while maintaining the original aspect ratio of the SWF file.</p> <p>Set to <code>exactfit</code> to make the entire SWF file visible in the specified area without trying to preserve the original aspect ratio. Distortion may occur.</p> |
| <code>wmode</code> | String | <p>Sets the Window Mode property of the SWF file for transparency, layering, and positioning in the browser. Valid values of <code>wmode</code> are <code>window</code>, <code>opaque</code>, and <code>transparent</code>.</p> <p>Set to <code>window</code> to play the SWF in its own rectangular window on a web page.</p> <p>Set to <code>opaque</code> to hide everything on the page behind it.</p> <p>Set to <code>transparent</code> so that the background of the HTML page shows through all transparent portions of the SWF file. This may slow animation performance.</p> <p>This property is not supported in all browsers and platforms.</p> |

Unsupported properties

The following optional Flash Player properties have no effect when used with Flex:

- `loop`
- `play`
- `swliveconnect`

Defining Flash MIME types

When your files are accessed from a web server, the server must properly identify them as Flash content in order to return them with the proper MIME type setting. You may be required to add the Flash SWF file MIME types to the server's configuration files and associate the following MIME types with the SWF file extensions:

- MIME type `application/x-shockwave-flash` has the `.swf` file extension.
- MIME type `application/futuresplash` has the `.spl` file extension.

Flash Player detection and deployment

Flex supports Flash Player version detection and deployment. When a user accesses your application, they are initially directed to a SWF file that detects the Flash Player version. If they have the specified version or later, the SWF file redirects the user to your content file, and your SWF file plays as designed. This process is transparent to the user.

If users don't have the specified version, they're redirected to an alternate location to download an updated Flash Player. This location is specified in the `flex-config.xml` file. The actual location depends on the type of player they are upgrading.

Note: If the user updates their version of Flash Player, they must restart their web browser.

The Flash Player detection logic is built into an HTML wrapper that is not visible under normal circumstances. If you write a custom wrapper for your Flex applications, you must manually add version detection.

When using the Flex JSP Tag Library to define a Flex application, Flex does not include the complete Flash Player detection scheme. However, the `<object>` and `<embed>` tags do include minimum version information and download locations.

Enterprise users who do not allow access outside of the corporate firewall can trigger Flash Player to go to a local detection and deployment page. For more information, see [“About the Enterprise Deployment Kit” on page 856](#).

Configuring detection and deployment

Version detection and deployment settings are defined in the `flex-config.xml` file. You use this file to configure version detection and deployment, or you can write your own custom version detection and deployment into the HTML wrapper.

Configuring version detection

The default minimum version for Flex applications is 7.0.14. This indicates version 7, major revision 0, minor revision 14. You can change the minimum version required in the `<flash-player>` tag of the `flex-config.xml` configuration file.

The following example shows the default version settings:

```
<flash-player>
  <required-version>7</required-version>
  <required-major-revision>0</required-major-revision>
  <required-minor-revision>14</required-minor-revision>
  ...
</flash-player>
```

Disabling version detection

You can disable Flash Player detection by setting the `<enable>` child tag to false. In closed environments where users cannot change their system configurations, disabling Flash Player detection provides a slight decrease in application start-up time.

The following example enables version detection:

```
<flash-player>
  <enable>true</enable>
  ...
</flash-player>
```

Configuring deployment settings

You can configure the deployment pages for the stand-alone Flash Player, ActiveX Player, and Netscape Plug-in in the `<flash-player>` child tags of the `flex-config.xml` configuration file. These tags can take the variable `{context.root}` as part of their value. This resolves to the root of the Flex application. Requests to `/flex` map to this root by default. The actual location of the context root depends on your application server.

When no Flash Player is installed, Flash uses the values in `<activex-download-url>` and `<plugin-download-url>`. These map directly to values in the `<object>` and `<embed>` tags in the HTML wrapper.

The `<object>` HTML tag supports versioning. When `<windows-auto-install>` is set to `true`, then the `<object>` tag checks the version information and calls the `<activex-download-url>` for an update, if required.

The following table describes the `<flash-player>` child tags:

| Tag | Description |
|-----------------------------------|---|
| <code>download-url</code> | The <code><download-url></code> is encountered when the client has Flash installed but doesn't have the correct version. The default location sends users to the Macromedia website so that they can download the latest version of Flash. If <code><windows-auto-install></code> is set to <code>true</code> , Flash ignores the <code><download-url></code> and directs Windows/IE clients to the <code><activex-download-url></code> . The default page specified by <code><download-url></code> is not visible on the file system but is generated by Flex. |
| <code>activex-download-url</code> | If the user has <code><windows-auto-install></code> set to <code>true</code> , Flash directs users to update the ActiveX version of Flash Player using this value. |
| <code>plugin-download-url</code> | Flex redirects users without the required Netscape Plug-in to this Plug-in upgrade page. |

The following example shows the default deployment settings in the `flex-config.xml` file:

```
<flash-player>
  <download-url>{context.root}/flex-internal/detection-kit/upgrade_flash
  /upgrade_flash.html</download-url>
  <activex-download-url>http://download.macromedia.com/pub/shockwave
  /cabs/flash/swflash.cab</download-url>
  <plugin-download-url>http://www.macromedia.com/go
  /getflashplayer</download-url>
  ...
</flash-player>
```

Each of the `<flash-player>` settings has a corresponding `https` setting. If the application server determines that the request is secure (if the request was made using the HTTPS protocol), Flex uses the `https` URLs in the HTML wrapper instead. You can disable the `https` settings by commenting them out in the `flex-config.xml` file.

Adding detection and deployment to custom wrappers

A Flash Player detection wrapper is sent prior to each MXML request, even before the default HTML wrapper. If you create a custom HTML wrapper, you must manually add version detection code if you want to support it.

The following example shows default detection code defined in the initial, invisible HTML wrapper. To add detection logic to your HTML wrapper, copy and paste this code into your HTML file.

```
<object id="flashDetection" classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000" codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version=4,0,0,0" width="80" height="80">

  <param name="src" value="/flex/flex-internal/detection-kit/flash_detection.swf?flashContentURL=%2Fflex%2Fwrapper%2Ffile1%2Emxml&altContentURL=%2Fflex%2Fflex%2Dinternal%2Fdetection%2Dkit%2Fupgrade%5Fflash%2Fupgrade%5Fflash%2Ehtml&contentVersion=7&contentMajorRevision=0&contentMinorRevision=14&allowFlashAutoInstall=true">
  <param name="quality" value="low">
  <param name="profile" value="false">

  <embed name="flashDetection" src="/flex/flex-internal/detection-kit/flash_detection.swf?flashContentURL=%2Fflex%2Fwrapper%2Ffile1%2Emxml&altContentURL=%2Fflex%2Fflex%2Dinternal%2Fdetection%2Dkit%2Fupgrade%5Fflash%2Fupgrade%5Fflash%2Ehtml&contentVersion=7&contentMajorRevision=0&contentMinorRevision=14&allowFlashAutoInstall=false" quality="low" profile="false" swliveconnect="true" pluginspage="http://www.macromedia.com/go/getflashplayer" type="application/x-shockwave-flash" width="80" height="80">
  </embed>

</object>
```

The version detection SWF file (*flash_detection.swf*) is located in the *flex_app_root/WEB-INF/lib/flex-bootstrap.jar* file. It is defined by the *BootstrapServlet* entry in the *flex-config.xml* file. The detection wrapper passes the version settings in the *flex-config.xml* file to the version detection SWF, as well as the deployment URLs if Flash Player requires an update.

To view the version detection wrapper, you can use a packet sniffer such as *TCPMonitor*. This application is included with *JRun*.

If you set `<windows-auto-install>` in the *flex-config.xml* file to `true`, Flash Player compares its version with the version defined in the `<object>` tag of the HTML wrapper rather than send a request to the version detection SWF. In this case, Flex does not generate a Flash Player detection wrapper.

About the Enterprise Deployment Kit

When a client detects that Flash Player is not installed or must be upgraded, the deployment settings in the *flex-config.xml* file guide the user to the correct location on the web to download a new player. If you are serving Flex applications from within a firewall, you can still provide users with upgrades to Flash Player using the Flex Enterprise Deployment Kit.

To distribute Flash Player you must provide a web page that guides your users to upgrade their player without ever going to an external website. In addition, you must store the necessary versions of Flash Player in an accessible location.

The Enterprise Deployment Kit includes all versions of Flash Player, plus a player version detection and download page. This page performs version, browser, and operating system using JavaScript. It guides users through the player upgrade and installation processes.

The Enterprise Deployment Kit includes the deployable versions of Flash Player and a player download page. The deployment kit requires that you agree to special licensing for Flash Player distribution. The license, implementation instructions, and download for the Enterprise Deployment Kit are available from:

http://www.macromedia.com/go/flex_deploy_kit

Optimizing version detection

Flash includes the following methods of short-circuiting version detection so that only the initial request for a Flex application performs it:

- **Internet Explorer (Windows)** If you set `<windows-auto-install>` in the `flex-config.xml` file to `true`, Flash Player compares its version with the version defined in the `<object>` tag of the HTML wrapper rather than send a request to the version detection SWF. If the versions do not match, users are redirected to the appropriate update page. The default value of `<windows-auto-install>` is `true`.
- **Netscape** Flash Player appends `?versionChecked=true` to the query string after the initial version detection succeeds. Subsequent requests do not trigger a new round trip request to the version detection SWF file.

Note: Netscape users who do not initially have any version of Flash Plug-in installed may be redirected to a Netscape download page rather than the Flash Player upgrade page. To avoid this, instruct your users to select `Edit > Preferences > Navigator > Helper Applications` and deselect the `Always Use the Netscape Plug-in Finder Service to Get Plug-ins` setting. If this setting is selected, Netscape ignores download location specified by the `<embed>` tag in the HTML wrapper.

Managing Flash Player auto-update

Flash Player supports auto-updating itself by periodically checking for new versions of the player on the `macromedia.com` site. IT administrators can customize the parameters of this update.

The auto-update settings can be configured in two ways:

- User settings in Flash Player
- `mms.cfg` file in user's "home" directory

Users can disable the auto-update process or set the periodicity of the checks by using the properties panel in Flash Player. These user-configured auto-update settings are stored in a local shared object.

The second method for configuring the auto-update settings is to create a file named `mms.cfg`. The `mms.cfg` file is intended to be configured by an IT administrator and stored on the user's computer. The file is not created by Flash Player installation. You might use a third-party administration tools, such as Microsoft System Management Server, to replicate the configuration file to the user's desktop.

1. Store the `mms.cfg` file in the following location, depending on your operating system:

- **Windows NT, 2K** C:/WINNT/System32
- **Windows XP** C:/WINDOWS/System32
- **Windows 95, 98, or ME** C:/Windows/System
- **Macintosh** /Application Support/Macromedia

The format of the `mms.cfg` file is a series of name=value pairs separated by carriage returns. If a parameter is not set in the file, Flash Player assumes the default value. When set, values in this file override the user-configured settings.

The following table describes settings in the `mms.cfg` file:

| Parameter | Default | Description |
|-------------------------------------|----------------|---|
| <code>AutoUpdateDisable</code> | 0 | 0 allows auto-update based on user settings. 1 disables auto-update. |
| <code>AutoUpdateInstallerUrl</code> | absent | String specifies URL as download location for player update. If this parameter is not set, Flash Player uses the Macromedia server. |
| <code>AutoUpdateInterval</code> | <0 (or absent) | <0 (or absent) uses value from player settings. 0 checks for updates every time the player launches. >0 specifies the minimum number of days between check for updates. |
| <code>AutoUpdateSettingsUrl</code> | absent | String specifies URL as destination for "Settings..." button in auto-update dialog. If this parameter is not set, Flash Player uses the Macromedia server. |
| <code>AutoUpdateVersionUrl</code> | absent | Specifies the URL to retrieve XML file containing Flash Player update data. If this parameter is not set, the player uses the Macromedia server. |

To disable the auto-update:

1. Open the `mms.cfg` file in a text editor.
2. Add the following auto-update setting:

```
AutoUpdateDisable=1
```

3. Save the `mms.cfg` file.
4. Close and restart Flash Player or the browser in which Flash Player is running.

You are not required to remove any other settings so that you can re-enable the auto-update feature and restore your original settings by removing this line (or setting its value to 0).

Passing request data to Flex applications

You can pass variables to Flex applications using a query string parameter or defining `flashVars` properties. When you use query string parameters, Flex converts them to `flashVars` variables and passes them into the application.

Flex does not recompile the application if the request variables have changed. As a result, if you dynamically set the values of the `flashVars` or query string parameters, you do not force a recompilation.

Parameters must be URL encoded. The format of the string is a set of name-value pairs separated by an ampersand (&). You can escape special and/or nonprintable characters with a percent sign (%) followed by a two-digit hexadecimal value. You can represent a single blank space using the plus sign (+).

The encoding for `flashVars` and query string parameters is the same as the page containing it. Internet Explorer provides UTF-16-compliant strings on the Windows platform. Netscape sends a UTF-8-encoded string to Flash Player.

Most browsers support a `flashVars` String or query string up to 64KB (65535 bytes) in length. They can include any number of name-value pairs.

To use the values of the passed Strings in a Flex application, you must declare them as variables. If you pass a query string parameter or `flashVars` property to Flex but do not declare it as a variable in the MXML file, Flex ignores the value.

To use `flashvars` or query string parameters inside your MXML file, declare the parameter name at the top of a script block. You can then access it as you would any variable inside the Flex application. You can bind its value to a display control or modify the value and return it to another function.

The following example defines the `name` and `hometown` variables and binds them to the text of `TextInput` controls:

```
<mx:Application xmlns:mx="http://www.macromedia.com/2003/mxml" >
  <mx:Script><![CDATA[
    var name: String = "None"; // Set default value if none is passed.
    var hometown: String = "None"; // Set default value if none is passed.
  ]]></mx:Script>
  <mx:TextInput text="Name: {name}" />
  <mx:TextInput text="Hometown: {hometown}" />
</mx:Application>
```

When a user requests this application with `name` and `hometown` parameters defined, Flex displays them in the `TextInput` controls. The following URL passes the name Reiner and the hometown Berlin to the Flex application:

`http://localhost:8100/flex/myApp.mxml?name=Reiner&hometown=Berlin`

Using query string parameters

You can add query string parameters to the client's request string, and interpret those parameters in almost any Flex application. In effect, Flex supports GET request variables passed in as name-value pairs in the URL.

When creating the HTML wrapper, Flex converts query string parameters to a `flashVars` variable. For example, if you use the following query string:

```
http://localhost:8101/flex/charts/PieChart1.mxml?fname=Reiner&lname=Knizia
```

Flex adds the following to the HTML wrapper:

```
<object ...>
  <param name='flashvars' value='lastname=Knizia&firstname=Reiner'>
  <embed ... flashvars='lastname=Knizia&firstname=Reiner' />
</object>
```

Flex cannot access data passed in using POST requests. You cannot pass query string parameters to a Flex application that runs inside a stand-alone Flash Player.

Using flashVars

You can pass variables to your Flex applications using the `flashVars` property in the `<object>` and `<embed>` tags in your HTML wrapper. You do this either when you write your own custom HTML wrapper or when you use the Flex JSP Tag Library's `<flash>` and `<mxml>` tags to generate the HTML output that displays the Flex application.

The following example sets the values of the `firstname`, `middlename`, and `lastname` `flashVars` properties:

```
<object classid='clsid:D27CDB6E-AE6D-11cf-96B8-444553540000' codebase='http://
download.macromedia.com/pub/shockwave/cabs/flash/
swflash.cab#version=7,0,14,0' width='500' height='500'>
  <param name='flashVars'
  value='firstname=Reiner&middlename=T&lastname=Knizia'>
  <param name='src' value='vartest.mxml.swf'>
  <embed pluginspage='http://www.macromedia.com/go/getflashplayer'
  width='500' height='500' flashVars='' src='vartest.mxml.swf' />
</object>
```

To pass `flashVars` to a Flex application in a JSP, add the `<flashvar>` tag that defines the `flashVars` properties to the `<flash>` or `<mxml>` tag, as the following example shows:

```
<%@ taglib uri="FlexTagLib" prefix="mm" %>
<h2>FlashVars Parameters</h2>
<mm:flash source="jsptest2.swf" supportembed="false" border="5">
  <mm:flashvar name="firstname" value="Reiner" />
  <mm:flashvar name="middlename" value="T" />
  <mm:flashvar name="lastname" value="Knizia" />
</mm:flash>
```

PART VI

Custom Components

This part describes how to create custom components for Macromedia Flex using Flash MX 2004.

The following chapter is included:

[Chapter 39: Using Custom Flex 1.0 Skins and Components in Flex 1.5 863](#)

CHAPTER 39

Using Custom Flex 1.0 Skins and Components in Flex 1.5

This chapter describes the changes to the workflow for creating custom components and skins using the flexforflash.zip file.

Contents

| | |
|---|-----|
| Overview | 863 |
| Skinning in Flash | 863 |
| Creating custom components in Flash | 864 |
| Updating existing components. | 864 |

Overview

Use of the flashforflex.zip file has been deprecated in Flex 1.5. A new version of the flexforflash.zip file is included in Flex 1.5 so that you can update existing components and skins from Flex 1.0 pre-Updater 2 to Flex 1.5. However, you should no longer create custom components in Flash for Flex using the flexforflash.zip file.

You can now skin components programmatically using ActionScript files. Programmatic skinning lets developers define skins with code rather than in the Flash environment, and it is more efficient and developer-friendly. It is also easier to create skins that are styleable using programmatic skinning.

You can also skin Flex components in Flash by using new sample skin files. Flex 1.5 includes two theme files called Pulse that you use as a basis for creating custom skins for your components.

Skinning in Flash

Skinning Flex components was formerly done entirely in the Flash environment. You could edit component skins in the flexforflash.zip file and export the FLA file as a SWC file. You would then use that SWC file as a theme in your Flex applications.

Now, you can programmatically skin most Flex components using ActionScript. You compile the ActionScript files into SWC files, which you use as themes in your Flex applications. For more information about programmatic skinning, see [“Programmatic skinning concepts” on page 438](#).

In addition to programmatic skinning, you can also reskin Flex components by using the sample Pulse themes included in Flex 1.5. These files are located in the *flex_install_dir/resources/themes/graphics* directory. For more information about graphical skinning, see [“Graphical skinning” on page 449](#).

Do not use the contents of the *flexforflash.zip* file to reskin components for Flex 1.5. The use of *flexforflash.zip* for reskinning Flex components has been deprecated.

Creating custom components in Flash

In Flex 1.0, you could create custom components for Flex in the Flash development environment. You modified the components in the *flexforflash.zip* file, and then exported them as SWC files for use in your Flex applications. This workflow has been deprecated for Flex 1.5.

You can still create components for Flex using the Flash environment, but Macromedia supports only creating components using the functionality supported in Flash for Flex 1.0 (including Updaters 1 and 2).

Do not create new components using the *flexforflash.zip* file for Flex 1.5 in Flash. This functionality has been deprecated. The *flexforflash.zip* file is provided for backward-compatibility only.

Updating existing components

Custom components that you compiled with the Flex 1.0 Updater 2 *flexforflash.zip* file will work in Flex 1.5. If you have components that you created for Flex 1.0 pre-Updater 2, you must update them before you can use them with Flex 1.5. You do this using the new version of the *flexforflash.zip* file that is included with Flex 1.5.

You delete the older *flexforflash.zip* file contents and extract the contents of the new *flexforflash.zip* file. You then open your components in Flash and export them as SWC files again. The new *flexforflash.zip* file is located in the *flex_install_dir/resources/flexforflash* directory.

To update components for use in Flex 1.5:

1. Close the Macromedia Flash 2004 environment.
2. Delete the existing *flexforflash* directories in Local Settings. The following directories are the default locations:
 - C:\Documents and Settings*user_name*\Local Settings\Application Data\Macromedia\Flash MX 2004\en\Configuration\Flex Classes\
 - C:\Documents and Settings*user_name*\Local Settings\Application Data\Macromedia\Flash MX 2004\en\Configuration\Components\Flex Components
 - C:\Documents and Settings*user_name*\Local Settings\Application Data\Macromedia\Flash MX 2004\en\Configuration\Classes\aso

3. Delete the existing FlexForFlash directories in the Flash environment. The following directories are the default locations:
 - C:\Program Files\Macromedia\Flash MX 2004\en\First Run\Flex Classes
 - C:\Program Files\Macromedia\Flash MX 2004\en\First Run\Components\Flex Components
4. Find the new flexforflash.zip file. It is located in the *flex_install_dir/resources/flexforflash* directory. Extract the contents of this file into your Flash First Run directory. The default location of this directory is C:\Program Files\Macromedia\Flash MX 2004\en\First Run.
5. Start the Flash environment.

Flash copies the contents of the First Run directory to the Local Settings directories.
6. Open your custom component's FLA file.
7. Ensure that your ActionScript classpath points to \$(LocalData)/Flex Classes. To check classpath settings, select File > Publish Settings.
8. Save the FLA file.

You must save the FLA file before exporting so that Flash updates the new classpath settings internally.
9. Export the SWC file.

For more information about working with custom components in the Flash environment, see www.macromedia.com/go/flex15_flexforflash.

INDEX

Symbols

@Embed tag, image import and 283
@font-face rule 424, 425

A

accessibility

- default reading and tab order 623
- enabling in Flex 816
- Flash Player and 619
- for hearing impaired users 625
- keyboard navigation for 625
- Macromedia Flash Accessibility web page 618, 620
- opaque and transparent windowless modes 619
- screen readers, configuring 619
- testing content 625

Accordion container

- Button controls and 265
- child initialization order 266
- default properties 264
- example 264
- keyboard navigation 265
- navigating panels 263
- navigation events 266
- resizing rules 264
- ToolTips 540

Accordion tag, syntax 267

ActionScript

- accessibility properties 624
- analyzing source 778
- asynchronous execution 670
- component configuration 20
- components 320
- custom event dispatcher 377
- defining components 359
- drawing API 444
- embedding images in 293

- errors 750

- events 335

- read-only properties 21

ActionScript components 373

ActionScript Profiler. *See* Profiler

activex-download-url 855

addEventListener() method 340

administering, flex-config.xml 806

Alert control

- events and 198

- icons for 199

- styles 200, 416

Alert pop-up. *See* Alert control

alignment, SWF files 852

allowscriptaccess option 849

annotationElements 519

anti-aliasing 852

Application class, alert() method 197

Application container

- about 191

- defaults 191

- filling 193

- sizing 171, 192

- styles 193

Application object, application property 331

Application tag

- rs: attribute 604

- syntax 195

- theme property 432

applications

- creating 836

- displaying in browser 845

- improving start time 578

- in MXML components 313

- root directory 799

- scaling 853

- scope 327

- securing 823
 - virtual directories 817
 - WEB-INF directory 837
- archive option 849
- AreaChart control 487
- AreaSeries 488
- argument binding 662
- asprofile query string 811
- AssetRenderer 512
- asynchronous functions, definition 779
- authentication
 - configuring 689
 - HTTP services 693
 - remote object services 689
 - web services 693
- auto-update parameters, Flash Player 858
- axes
 - formatting lines 508
 - rotating labels 506
 - titles 504
- AxisRenderer
 - about 502
 - formatting axis lines 508
 - rotating axis labels 506
 - tick marks 507
 - titles 504

B

- background color 406, 852
- backgroundElements 519
- BarChart control
 - about 489
 - customizing 536
- BarSeries 489
- batch files, compiling 797
- beginFill 445
- beginGradientFill 445
- behaviors
 - in applications 453
 - compositing effects 465
 - custom effect triggers 472
 - custom effects 465
 - customizing effects 464
 - defining custom effects 466, 468
 - effect triggers 454
 - effects in ActionScript 471
 - standard effects 460
 - using 24
 - ViewStack container and 252
- best practices, security 832

- binding
 - complex results 667
 - data 637
 - service result objects 665
 - showing warnings 811
 - styles 406
- borders, skinning 443
- borderStyle property, determining state 440
- Box container
 - default properties 209
 - example 209
 - sizing 210
- breakpoints
 - setting 759
- browser, back and forward commands 569
- BubbleChart controls 490
- BubbleSeries 490
- Button control
 - about 37
 - example 38
 - sizing 38
 - user interaction 39
- Button tag, syntax 40
- buttonStyleDeclaration property 416

C

- cache-sws 808
- caching
 - custom components 808
 - fonts 809
- Canvas container
 - and the Image control 288
 - example 208
 - layout 206
- Canvas tag, syntax 209
- CartesianChart control 536
- Cascading Style Sheets. *See* CSS
- CategoryAxis
 - about 480, 481
 - dataProvider 482
- cell renderer
 - about 379
 - API 380
 - creating 379
 - examples 382
- centimeters, style property 398
- ChangeEvent metadata keyword 369

charts

- about 475
- AreaChart control 487
- AreaSeries 488
- AssetRenderer 512
- axes 480
- axis labels 505
- axis styles 501
- axis titles 504
- AxisRenderer 481, 502
- background colors 514
- BarChart control 489
- BarSeries 489
- BubbleChart controls 490
- BubbleSeries 490
- CartesianChart control 536
- CategoryAxis 481
- ColumnChart control 491
- ColumnSeries 491
- controls 477
- custom 536
- dataProvider property 482
- DataTips 509
- defining data 483
- disabling events 530
- doughnut charts 494, 498
- effects 531
- events 527
- fills 514
- finding data points 529
- grid lines 519
- HitData 527
- inline styles 502
- Legend controls 521
- LegendItem objects 523
- LinearAxis 481
- LineChart control 492
- LineSeries 492
- margins 502
- minimum values 512
- mouse sensitivity 529
- PieChart control 494
- PieChart control labels 496
- PieSeries 495
- PlotChart control 498
- PlotSeries 499
- renderer interfaces 479
- renderers 478
- rotating axis elements 506
- stacking 526

- styles with CSS 501
- tick marks 482, 507
- types 477
- using images 512
- using multiple data series 538

CheckBox control

- about 40
- example 41
- reskinning 433
- user interaction 41

CheckBox tag, syntax 42

- child component 163

- child controls, deferring creation 577

- child descriptor, definition 584

childDescriptors property

- about 584
- id 585
- properties 586
- type 586

class selectors

- about 403
- charts 501

classConstruct() method 442

classes

- createClassObject() method 367
- hierarchy for components 16
- MovieClip 17
- scope 329
- securing 825
- UIComponent properties 18
- UIObject properties 18
- See also* specific class name 367

classid 848

classpath

- compiling 799
- custom components 373

clear() method

- calling 441
- description 445

- client. *See* Flash Player

- clustered charts 526

- codebase 848

- codetype option 850

- Color style format, style property 399

ColumnChart control

- about 491
- customizing 536

ColumnSeries 491

- ComboBox control
 - change event for 144
 - data provider 143
 - editable 141
 - event handling 142
 - example 142
 - keyboard navigation 145
 - user interaction 145
- ComboBox tag, syntax 146
- compc utility
 - about 800, 801
 - examples 803
- compiler, Application tag options 195
- compiling
 - accessibility 797
 - with compc utility 448
 - debug password 797, 812
 - with mxmle 448
 - optimizing 799
 - Profiler 799
 - programmatic skins 448
 - proxy URL 799
 - RSLs 803
 - showing stack traces 812
 - showing warnings 811
 - SWC files 800
 - symbols 815
- component layout
 - at runtime 179
 - configuring 171
- component sizing
 - layout pass 170
 - resizing 171
 - rules 171
- components
 - about 15
 - accessibility and 621
 - adding to RSLs 600
 - calling methods in ActionScript 320
 - class hierarchy 16
 - common properties 18
 - compiling 800
 - compound 366
 - configuring 28
 - containers 161
 - controls 32
 - creating 367
 - creating in Flash 863
 - creating, MXML 311
 - destroying 582
 - distributing 806, 842
 - dynamic 269
 - emitting events 365
 - explicit size 172
 - gaps between 173
 - getters and setters 361
 - initialize event 21
 - initialize method 321
 - manifest files 805
 - namespaces 804
 - preferred size 172
 - properties 322
 - registering events 344
 - sizing 26, 170
 - startup order 348
 - using 15
 - using Alert function 331
 - zooming 458
- compound components 366
- compound selectors 404
- configuring
 - ActionScript classpath 814
 - ActionScript optimizer 813
 - debugger 754
 - debugging 841
 - fonts 427
 - logging 819
 - Profiler 814
 - web.xml file 838
- console settings 820
- Container class, syntax 166
- containers
 - about 161
 - Accordion 263
 - Application 191
 - behaviors 189
 - Box 209
 - Canvas 206
 - class hierarchy 166
 - common properties 186
 - configuring 185
 - ControlBar 211
 - createComponent() method 582
 - createComponents() method 583
 - createLater() method 589
 - creating children 182
 - creating MXML components 312
 - creation order 587
 - DividedBox 212
 - effects and 456

- enabling 168, 550
- events 188
- example 166
- Form 215
- Grid 229
- HBox 209
- HDividedBox 212
- initialize event 580
- layout 205
- LinkBar 253
- navigator 247
- Panel 168
- setChildIndex() method 185
- setting creationPolicy 578
- sizing 171
- skins 189
- style inheritance 408
- styles 187
- TabBar 255
- TabNavigator 259
- Tile 237
- TitleWindow 239
- VBox 209
- VDividedBox 212
- ViewStack 248
- content area, sizing 162
- content-size 808
- content-type 850
- context, function scoping 325
- ControlBar container
 - default properties 211
 - example 211
- ControlBar tag, syntax 212
- controls
 - about 32
 - Alert 197
 - appearance 37
 - Button 37
 - CheckBox 40
 - class hierarchy 35
 - ComboBox 141
 - creating MXML components 311
 - DataGrid 127
 - DateChooser 42
 - DateField 49
 - deferring instantiation 578
 - FormHeading 216
 - history, managing 569
 - HorizontalList 116
 - HRule 53

- HScrollBar 81
- HSlider 56
- Image 282
- Label 63
- Link 68
- List 105
- Loader 70
- MediaController 297
- MediaDisplay 297
- MediaPlayback 297
- Menu 148
- MenuBar 153
- NumericStepper 72
- positioning 37
- ProgressBar 74
- RadioButton 77
- ScrollBar 81
- sizing 36
- Spacer 178
- Text 82
- TextArea 85
- TextInput 88
- TileList 121
- Tree 135
- using 31
- VRule 53
- VScrollBar 81
- VSlider 56
- See also* components
- create() method 326
- create-compile-report 746
- createChild() method 323
- createChildren method 367
- createClassObject method 367
- createComponent() method 582
- createComponents() method 583
- createLater() method 589
- creating components
 - ActionScript 323
 - numChildren property 185
 - using metadata statements 368
- creationComplete event 348, 355
- creationIndex property 589
- creationPolicy property
 - about 589
 - definition 578
 - setting queue order 589
- credit cards, validating 713
- cross-domain-policy tag 830

CSS

- Application type selector 414
- charting 501
- colors 400
- embedded resources 407
- external 412
- global style sheet 412
- global type selector 415
- inheritance 402
- supported properties 406
- CSSStyleDeclaration class 418
- curly braces, data binding 638
- currency
 - formatting 728
 - validating 712
- currentToolTip 543
- Cursor Manager
 - example 548
 - SWF files 549
 - using 547
- CursorManager class
 - setCursor() method 548
 - syntax 551
- cursors
 - busy 549
 - creating and removing 548
 - file types for 547
 - priority 547
 - wait 548
- curveTo 445
- custom components
 - classpath 373
 - compound 366
 - events 363
 - example 360
 - faceless 375
 - metadata 368
 - mouse events 364
 - namespace 374
 - organizing 373
 - passing data 362
 - types 361
- custom tags
 - taglib definition 840
 - See also* JSP tag library
- custom, formatting 738

D

- data
 - binding 631
 - Binding tag 639
 - binding, curly braces 638
 - data service results 665
 - formatters, custom 738
 - formatters, standard 726
 - formatting 725
 - formatting in List-based controls 391
 - models 647
 - results, data service 665
 - See also* data validation
 - session 681
 - sources, accessing 634
 - types, converting 676, 680
 - validating in List-based controls 390
 - validators, standard 712
- data models
 - as value object 652
 - form data and 223
 - validating data 651
- data provider API
 - for components 93
 - for controls 95
 - using 94
- data providers
 - array 92
 - controls for 91
 - custom 96
 - data models and 96, 98
 - DataProvider interface 92
 - list-based 92
 - MXML syntax 97
 - structure 92, 483
 - TreeNode class and 92
 - using 91, 92
- data services
 - authentication 689
 - callback URLs 672
 - calling 659
 - common properties 697
 - debugging 673
 - declaring 658
 - diagram 656
 - Flex proxy 662, 664
 - HTTP service properties 699
 - remote object service properties 698
 - results 665
 - securing 687

- web service properties 698
- whitelists 687, 701
- data validation
 - about 632, 703
 - custom 706
 - form 707
 - models 651
 - programmatic 704
- data validators, standard 712
- data visualization 475
- DataGrid control
 - binding limitation 130
 - column order 128
 - column properties 129
 - custom header 384
 - data provider 128
 - DataGridColumn tag and 129
 - example 127
 - keyboard navigation 132
 - populating 129
 - printing 610
 - selected item 130
 - user interaction 131
- DataGrid tag, syntax 132
- DataGridColumn tag, syntax 134
- dataProvider property
 - charts 482, 483
- DataTips 108, 509
- Date class
 - DateChooser control and 42
 - DateField control and 49
- DateChooser control
 - ActionScript and 45
 - example 44
 - user interaction 46
- DateChooser tag, syntax 47
- DateControl control, styles 416
- DateField control
 - about 49
 - date formatter function 50
 - DateChooser and 49
 - example 49
- dates
 - formatting 734
 - validating 715
- debug files, generating 798
- debugger
 - common commands 757
 - convenience variables 763
 - example 767
 - setting default browser 755
 - status 765
 - using breakpoints 759
 - using watchpoints 761
- debugging
 - ActionScript files 752
 - configuring fdb 754
 - data services 673
 - generating SWF files 811
 - logging compiler errors 812
 - Profiler 814
 - programmatic skins 448
 - remote 756
 - showing source code 812
 - source command 761
 - SWD files 753
 - trace() function 748
 - using fdb debugger 752
 - web services proxy 811
- declarative security 824
- default browser
 - definition 755
 - setting 755
- default button
 - Form container 218
 - syntax 169
- deferred instantiation, example 333
- delay times 544
- Delegate.create() method 343
- deploying
 - about 835
 - components 840
 - context root 817
 - headless servers 816
 - programmatic skins 448
- deprecation 813
- destroying components 582
- detecting, Flash Player version 853
- Developer Edition 840
- device fonts 422
- DHTML, HTML wrapper 843
- Dissolve effect 460
- DividedBox container
 - default properties 213
 - dividers 214
 - events 214
 - example 213
 - live dragging 214
- Document object, scope 330
- doLater() method 333

- double-byte fonts 428
- doughnut charts 494, 498
- download progress bar
 - customizing 201
 - disabling 201
 - syntax 201
 - using 200
- download-url 855
- Drag and Drop Manager
 - containers as target 561
 - DataGrid control and 562
 - drag initiation 555
 - drag initiator 553
 - drag proxy 553
 - drag target 554
 - dragBegin event
 - description 554
 - dragEnter event
 - description 554
 - DragSource class and 556
 - example 557
 - operation 555
 - using 553
- Drag Manager, UIObject class syntax 567
- dragBegin event
 - description 355
 - Drag and Drop Manager and 554
 - handling 556
- dragComplete event
 - description 355
 - Drag and Drop Manager and 554
 - handling 561
- dragDrop event
 - description 355
 - Drag and Drop Manager and 554
 - handling 560
- dragEnter event
 - description 355
- dragEnter event, handling 560
- dragExit event
 - description 355
 - Drag and Drop Manager and 554
 - handling 560
- DragManager class
 - syntax 565
 - using 555
- dragOver event
 - description 355
 - Drag and Drop Manager and 554
 - handling 560
- DragSource class
 - syntax 566
 - using 555
- draw event 348, 355
- drawing 444
- duration, Time style format 399

E

- e-mail address, validating 717
- easing functions 466
- effect
 - containers and 456
 - defining custom 466
- Effect metadata keyword 369, 370
- effectEnd event 355
- effects
 - applying 454
 - charts 531
 - composite 465
 - custom 465
 - custom components 370
 - custom triggers 472
 - customizing 464
 - defining custom 468
 - defining in ActionScript 471
 - easing functions 466
 - Effect metadata keyword 370
 - events 455
 - layout updates and 179
 - queued containers 593
 - Resize 287
 - standard 460
 - with ToolTips 545
- effectStart event 355
- 8-bit octet RGB 400
- embed images in ActionScript 293
- Embed metadata keyword
 - description 369
 - import MP3 files 293, 298
- Embed style 407
- embed tag
 - default parameters 848
 - example 847
 - unsupported parameters 853
 - use in browsers 846
- embedded fonts
 - caching 426
 - identifying 425
 - isFontEmbedded() method 426
- embeddedfontlist property 425

- emitting events 365
- ems, style property 398
- enableAccessibility() method 816
- encoding 816, 859
- endFill 445
- Enterprise Deployment Kit 856
- environments, without Windows systems 798
- ERROR log level 818
- error page mappings 839
- error reporting
 - error messages 748
 - error types 748
 - FTP errors 752
 - HOMEPATH 749
 - HTTP errors 751
 - log file name 747
 - network errors 752
 - supported errors 750
- ErrorReportingEnable 750
- event dispatcher, example 377
- event handlers
 - about 336
 - inline 338
 - registering multiple 346
- event listeners
 - creating classes 347
 - inline 342
 - multiple 344
 - scope 341
- Event metadata keyword 351, 365, 369, 370
- event object
 - definition 336
 - properties 337
 - target property 341
- events
 - Application 353
 - base class 353
 - ChangeEvent metadata keyword 369
 - charts 527
 - custom components 363
 - Delegate class 343
 - dispatchEvent() method 351
 - emitting 365
 - Event metadata keyword 369, 370
 - handleEvent() method 346
 - handler 336
 - handling 25
 - HitData 527
 - initialize 363
 - manually dispatching 351
 - mouse 353
 - mouse events 352, 364
 - NonCommittingChangeEvent metadata keyword 372
 - explicit argument passing 660
 - expressions, JSPs 787
 - extending graphical components 366

F

- faceless components, definition 361
- Fade effect 461
- faults, handling 766
- fdb 752
- file encoding 816
- file-watcher-interval 809
- filepaths in flex-config.xml 807
- filling the Application container 193
- fills
 - alpha 518
 - charts 514
 - gradient fills 517
- findDataPoint() method 529
- firewalls 826
- Flash
 - data, managing 635
 - graphical skins 450
- Flash client. *See* Flash Player
- Flash Debug Player, stand-alone version 749
- Flash IDE. *See* Flash MX 2004
- Flash MX 2004
 - generating SWD files 776
- Flash Player
 - accessing files 829
 - auto-update parameters 858
 - configuring 772, 858
 - context menu 608
 - debug version 746
 - device fonts 422
 - embedded fonts 423
 - frame statistics 779
 - parameters 858
 - playback quality 852
 - security sandbox 825
 - upgrading 857
- flash tag 788
- flash_detection.swf 856
- flashvar tag 789
- flashVars
 - object and embed tags 848

- Flex
 - configuring 795
 - license key 840
 - proxy 662
- Flex applications
 - accessibility 617
 - printing 607
- Flex JSP tag library 782
- Flex proxy 662
 - securing 694
 - web and HTTP services 664
- flex-config.xml
 - accessible tag 816
 - actionscript-classpath tag 814
 - actionscript-file-encoding 817
 - cache 808
 - compiler 813
 - editing 806
 - fonts 809
 - global-css-url 412
 - headless-server 816
 - keep-generated-as 813
 - keep-generated-swfs 813
 - language-range 427
 - log-compiler-errors tag 820
 - optimize tag 814
 - production mode 807
 - securing named services 831
 - styles 809
 - system-classes tag 815
- flexforflash.zip file 863
- flexible sizing 26, 36, 173
- font-face
 - character ranges 427
 - example 424
- fontFamily style 422, 424
- fonts
 - @Embed tag, image import and 283
 - @font-face rule 424, 425
 - and Flash Player 422
 - configuring in flex-config.xml file 427
 - device 422
 - double-byte 428
 - embedded 423, 810
 - multiple faces 425
- fontSize style 399
- for in loops 323
- Form container
 - data services and 227
 - example 216
 - FormItem 217
 - populating data models from 225
 - required fields of 220
 - sizing and positioning children 220
 - spacing within 219
 - submitting data 226
 - form data
 - in controls 222
 - storing 221
 - Form tag, syntax 229
 - formatters, List-based controls and 391
 - formatting, data
 - about 633
 - currency 728
 - custom 738
 - dates 734
 - errors, handling 726
 - numbers 727
 - phone numbers 731
 - ZIP codes 733, 737
 - FormHeading tag, syntax 229
 - FormItem tag, syntax 229
 - frame rate 196
 - FrameProfilingEnable parameter 772
 - frames
 - delay 779
 - printing 608
 - frameworks 798
 - functions, queuing 333
 - futuresplash MIME type 853
- G**
 - gaps between components 173
 - gateway URL 798
 - generated files, saving 811, 812
 - GET requests 860
 - getters 361
 - global events 353
 - global styles 397
 - global.css 809
 - glyphs, character ranges 426
 - gradient fills, charts 517
 - graphical skinning 449
 - graphical skins 429
 - Grid container
 - children 229
 - column spacing 233
 - column span 232
 - columns, arranging 229
 - default properties 230

- example 230
 - sizing and positioning children 233
- grid lines 519
- Grid tag, syntax 234
- GridItem, syntax 234
- GridRow tag, syntax 234

H

- handleEvent() method 346
- handling events 337
- HBox container. *See* Box container
- HDividedBox container. *See* DividedBox container
- headless, java.awt 816
- help, ToolTips 539
- hexadecimal color format 399
- hideDataEffect trigger 532
- hierarchical data providers
 - considerations 105
 - objects for 104
 - XML data for 101
- history management
 - custom 571
 - standard 569
- History Manager, about 569
- HistoryManager class
 - methods 571
 - syntax 571
- HitData object
 - findDataPoint() method 529
 - using 527
- HOMEDRIVE 749
- HOMEPATH 749
- horizontalAxis 480
- horizontalAxisStyle 501
- horizontalFill 519
- HorizontalList control
 - examples 117
 - keyboard navigation 120
 - user interaction 120
- HorizontalList tag, syntax 120
- horizontalStroke 519
- hotspot 352
- HRule control, about 53
- HSlider control
 - about 56
 - events 58
 - example 57
 - keyboard navigation 60
 - labels 57
 - multiple thumbs 59

- slider thumb 57
- tick marks 57
- ToolTips 60
- track 57
- HSlider tag, syntax 61
- HTML text
 - anchor tags 65
 - bold tag 66
 - font tag 66
 - in Label control 64
 - italic tag 66
 - list tag 66
 - paragraph tag 67
 - span tag 67
 - tags enclosed in quotation marks 65
 - underline tag 67
 - See also* Label control
- HTML wrapper
 - adding default 843
 - customizing 846
 - detection and deployment 856
 - display tags 849
 - example 843
 - parameters 849
- HTML, object and embed tags 846
- HTTP errors 751
- HTTP services
 - access control 688
 - authentication 693
 - debugging 812
 - properties 699
 - whitelists 688
- http-maximum-age 809
- http-service-proxy-debug 746
- HTTPS
 - deployment settings 855
 - security and 832
 - using in Flex 695

I

- icon function, using 111, 138
- IconFile metadata keyword 369
- Image control
 - file path 285
 - in a Canvas container 288
 - positioning 288
 - sizing 285
 - visibility 289

- image import
 - about 282
 - Image control and 282
 - PNG files 282
- Image tag, maintainAspectRatio 286
- images
 - adding to RSLs 601
 - charts 512
 - Embed metadata keyword 369
- inches, style property 398
- information, logging 818
- initialize event
 - about 363
 - components 363
 - for components 348
 - containers 580
 - instantiation order 581
 - for UIObject class 355
 - using 339
- initObject 368
- inline styles, overriding 420, 421
- innerRadius 498
- Inspectable metadata keyword 369, 371
- instantiating components 323
- instantiation
 - childrenCreated event 167
 - creating children in components 367
 - order 349
- introspection 323

J

- J2EE security 824
- JavaServer Pages. *See* JSP pages
- JNDI, accessing 675
- JSP pages
 - deploying 844
 - Flash Player settings 789
 - flash tag 788
 - flashvar tag 789
 - mxml tag 785
 - param tag 789
 - passing data in 789
 - query string parameters 791
 - writing MXML 785
- JSP tag library, flashVars parameter 860

K

- keyboard events 347
- keywords, metadata 369

L

- Label control
 - about 63
 - example 63
 - HTML text and 65
 - text property 64
- label function, using 107
- Label tag, syntax 67
- labelFunction 505
- labelPosition property, PieChart controls 496
- labels
 - axis 505
 - rotating 506
- lag time 779
- layout containers
 - about 162, 206
 - Box 209
 - Canvas 206
 - ControlBar 211
 - DividedBox 212
 - Form 215
 - Grid 229
 - Panel 168, 234
 - Tile 237
 - TitleWindow 239
 - See also* components
- layout pass 170
- layout performance 587
- layouts. *See* components
- Legend controls 521
- LegendItem object 523
- Length format
 - about 398
 - relative units 398
- lib-path element 815
- library descriptor file 597
- library file 597
- license
 - changing key 840
 - version 799
- licensetool utility 840
- LinearAxis 480, 481
- LineChart control
 - about 492
 - adding images 514
- LineSeries 492
- lineStyle 446
- lineTo 446

- Link control
 - about 68
 - example 68
 - user interaction 69
- Link tag, syntax 69
- LinkBar container
 - default properties 253
 - example 254
- LinkBar tag, syntax 255
- List control
 - cells, custom 379
 - data provider 96
 - DataTips 108
 - Drag and Drop Manager and 562
 - events 106
 - example 105
 - icon field 111
 - item index 106
 - keyboard navigation 112
 - label function 107
 - row colors 112
 - ScrollTips 108
 - sorting 110
 - user interaction 112
- list data providers
 - ActionScript and 99
 - arrays and 97
 - binding data to 98
- List tag
 - dragEnabled 562
 - syntax 113
- List-based controls
 - formatting data 379
 - validating data 379
- listeners, events 340
- Live Preview SWF files 800
- Loader control
 - about 70
 - example 70
 - image import 289
 - loading Flex applications 70
 - sizing 71
- Loader tag, syntax 71
- loading images 291
- loadPolicyFile() method 829
- LocalConnection object 827
- log files, using trace() function 748
- logging
 - errors 750
 - warning 750

M

- managers
 - Cursor 547
 - Drag and Drop 553
 - History Manager 569
 - PopUp Manager 243
 - ToolTip Manager 543
- manifest files 798, 805
- margins
 - Application container 173
 - charts 502
- master skin symbols 430
- MaxWarnings 750
- media controls, sizing 302
- media import
 - about 297
 - MP3 files 297
- MediaController control
 - about 300
 - using 300
- MediaDisplay control
 - about 299
 - using 299
- MediaPlayback control
 - about 302
 - cue points 302
 - using 302
- MediaPlayback tag, syntax 303
- Menu control
 - events 150
 - keyboard control 152
 - menu items 148
 - menu items, type 149
 - populating 148
 - separators for 150
 - syntax 152
 - XML data for 149
- menu item syntax 158
- MenuBar control
 - events 156
 - menu items, adding 155
 - user interaction 157
 - XML data for 153
- MenuBar tag, syntax 157
- messageStyleDeclaration 416
- metadata
 - in MXML components 316
 - inspectable properties 371
 - syntax 368
 - tags 369

- metadata keywords
 - available 369
 - ChangeEvent 369
 - custom components 368
 - Effect 370
 - Embed 369
 - Event 365, 370
 - Inspectable 371
 - NonCommittingChangeEvent 372
 - Style 373
 - using 368
- methods, allowDomain 828
- millimeters, style property 398
- contentType property 294
- minField 512
- minor tick marks 507
- minorTickPlacement property 507
- mm.cfg
 - syntax 750
 - using 749
- models, data 631
- mouse events
 - charts 527
 - handling 353, 364
- mouse pointer 352
- mouseChangeSomewhere 352
- mouseChangeSomewhere event 356
- mouseDown event 356
- mouseDownEvent, Drag and Drop Manager and 554
- mouseDownSomewhere 352
- mouseDownSomewhere event 356
- mouseMove event 356
- mouseMoveSomewhere event 352, 356
- mouseOut event
 - about 356
 - example 364
- mouseOver event
 - about 356
 - delay times for ToolTips 544
 - example 364
- mouseSensitivity property 529
- mouseUp event 356
- mouseUpSomewhere event 352, 356
- mouseX 352
- Move effect 462
- move event 357
- moveTo() method 446
- MovieClip class, syntax 17
- MSAA (Microsoft Active Accessibility) 619
- mx.controls.ToolTip class 541
- mx.managers.ToolTipManager class 543
- mx.styles.StyleManager 417
- mx:Application tag, Application object 329
- mx:Script tag, scope 329
- MXML
 - including files in JSPs 787
 - multiple files 309
- MXML components
 - custom properties and methods 313
 - distributing 842
 - referencing 310
- mx:tag 785
- mx:tag-size 808
- mx:tag tool 796

N

- namespace, custom components 374
- namespaces 804, 841
- navigator containers
 - about 247
 - Accordion 263
 - creationPolicy 579
 - definition 577
 - LinkBar 253
 - TabBar 255
 - TabNavigator 259
 - ViewStack 248
 - See also* components
- NonCommittingChangeEvent metadata keyword 369, 372
- numbers
 - formatting 727
 - validating 718
- NumericStepper control
 - about 72
 - example 72
 - keyboard navigation 73
 - sizing 73
 - user interaction 73
- NumericStepper tag, syntax 73

O

- object listeners 343
- object tag 846
- objects
 - Application 329
 - creating 582
 - listing properties 323
 - scope 326

- optimizing
 - performance 769
 - Windows auto-install 857
- overlaid charts 526

P

- packaged components 804
- Panel container
 - ControlBar and 236
 - default properties 235
 - example 235
 - styles 416
- Panel tag, syntax 237
- param tag 789
- parentApplication property 333
- Pause effect 462
- performance
 - caching 808
 - EJBs 769
 - layouts 587
 - RSLs 595
- permission, policy files 830
- phone numbers
 - formatting 731
 - validating 719
- picas, style property 398
- PieChart control
 - about 494
 - labels 496
- PieSeries
 - about 495
 - fills 515
- pixels, style property 398
- player. *See* Flash Player
- PlotChart control 498
- PlotSeries 499
- plug-in, deploying 855
- plugin-download-url 855
- pluginspage 849
- pointRenderer property 514
- points, style property 398
- policy files
 - example 831
 - syntax 830
- PopUp Manager
 - passing arguments to 244
 - TitleWindow container and 243
- postal codes, formatting 733, 737
- printers, supported 608

- printing
 - adding pages 613
 - bitmaps 615
 - controls 611
 - deleting print job 615
 - Flash Player context menu 608
 - print area 613
 - printers supported 608
 - root document 613
 - scaling 614
 - starting print job 612
- PrintJob class
 - addPage() method 613
 - creating 609
 - object and class 608
 - send() method 615
 - start() method 609
 - syntax 612
 - using 609
- production mode, debugging 810
- profile() method 773
- ProfileFunctionEnable 772
- Profiler
 - about 770
 - configuring 771
 - data directory 770
 - disabling 808
 - installing 770
 - SWD files 774
 - using 770
- profiler.war 770
- ProfilingOutputDirectory 772
- ProfilingOutputFileEnable 772
- programmatic security 824
- programmatic skinning 438
- programmatic skins 429
 - compiling 448
 - debugging 448
 - deploying 448
- progress bar and loading images 291
- ProgressBar control
 - about 74
 - example 75
 - labels 76
 - manual mode 74
- ProgressBar tag, syntax 76
- properties
 - inspectable 371
 - listing 323
- proto chain, debugging 767

- proxyallowurlovrider 799
- Pulse skins 430
- Pulse themes 449

Q

- query string parameters
 - about 859
 - special characters 859
 - using 860

R

- RadioButton control
 - about 77
 - example 78
 - user interaction 78
- RadioButton tag, syntax 80
- RadioBottonGroup control
 - about 79
 - tag, syntax 80
- RectBorder 443
- remote object services
 - access control 687
 - authentication 689
 - properties 698
 - stateful objects 675
 - stateless objects 674
 - whitelists 687
- remote objects
 - busy cursor and 551
 - debugging 812
- remote requests, security 831
- remote-objects-debug 746
- remoteallowurlovrider 799
- RemoteObject tag, showBusyCursor 551
- removeEventListener() method 341
- renderer, AssetRenderer 512
- Repeater object
 - properties 270
 - repeating MXML components 271
- repeating components 269
- request data
 - flashVars 860
 - passing 859
 - See also* query string parameters
- Resize effect 287, 463
- resize event 357
- resizing components 173
- RGB color format 400

- RSLs
 - compiling 803
 - defining with SWS file 597
- runtime shared libraries (RSLs), definition 595

S

- SampleRectBorder.as 443
- scope
 - about 325
 - Document object 330
 - event handlers 342
 - global request data 790
 - isDocument() method 332
 - parentDocument property 332
 - this keyword 328
- screen readers, detecting with ActionScript 624
- scroll bars
 - sizing 181
 - using 180
- ScrollBar control
 - about 81
 - example 81
 - sizing 81
 - user interaction 82
- ScrollBar tag, syntax 82
- ScrollTips 108
- security
 - accessing HTTPS resources 830
 - compatibility with Flash Player 832
 - data services 687
 - data transport 832
 - domains 826, 828
 - encryption 832
 - file access 827
 - Flash Player 825
 - J2EE 824
 - resources 833
 - services 824
 - SharedObjects 827
 - whitelist 824
- security sandbox, tunnelling 828
- selectors
 - class 24, 401
 - precedence 405
 - type 24, 402
- SeriesInterpolate effect 535
- SeriesSlide effect 534
- SeriesZoom effect 534

- services
 - configuring access to 687
 - data 631
- servlet definitions 838
- session data 681
- sessions, servlet 681
- setCursor() method 548
- setters, defining for custom components 361
- SharedObjects, security 826
- show event 350
- show-all-warnings 747
- show-binding-warnings 747
- show-override-warnings 747
- show-source-in-compiler-errors 747
- show-stacktraces-in-browser 747
- showDataEffect trigger 532
- showLine property 508
- skinning 429
 - borders 443
 - borderStyle property 440
 - classConstruct() method 442
 - clear() method 441
 - compiling programmatic skins 448
 - defining symbols 439
 - definition 25, 447
 - deploying programmatic skins 448
 - example 433
 - Flash 863
 - graphical skinning 449
 - master skin symbols 430
 - programmatic 438
 - styleable properties 447
 - symbolName property 430
 - using Flash 451
- slider controls 57
- SOAP, web services 656
- Social Security numbers, validating 720
- Spacer control 178
- Spacer tag, syntax 178
- src property 849
- SSL 832
- stack traces, debugger 765
- stacked charts 526
- startup
 - performance 587
 - time, RSLs 595
- strings, validating 721
- style inheritance, flowchart 409
- Style metadata keyword 369, 373
- style sheets. *See* CSS
- Style tag
 - external style sheets 412
 - local style definitions 413
- StyleManager class
 - about 396
 - global styles 417
- styles
 - about 395
 - binding 406
 - charting 501
 - compound components 415
 - inheritance 407
 - inline 421
 - plain 405
 - setStyle() method 419
 - skinning 447
 - Style metadata keyword 369, 373
 - Style tag 396
 - ToolTips 541
 - using 23
 - value formats 398
- SVG files, importing 284
- SWC files
 - about 800
 - compiling 801
 - distributing 806, 841
 - manifest files 805
 - RSLs 597
 - themes 432
- SWD files 753
 - compiling 796
 - generating 774
- SWF files
 - analyzing 776
 - deploying 843
 - HTTPS 826
 - importing 283
 - including in JSPs 788
 - keeping generated 811
 - MIME types 853
 - printing frames 608
 - security 826
 - symbol import 284
 - viewing 766
- SWO files, caching 808
- SWS file 597
- symbolName property 430
- symbols
 - adding to RSLs 602
 - definition 429

- reporting 812
- skinning 439
- syntax, for metadata statements 368
- system path, virtual directories 817
- System.security.loadPolicyFile() method 830

T

- tab order, overview 623

- TabBar container

- data initialization 257
 - data provider 96
 - default properties 256
 - events 258
 - example 256
 - ViewStack and 250

- TabBar tag, syntax 259

- TabNavigator container

- child initialization order 261
 - default properties 260
 - example 260
 - keyboard navigation 262
 - resizing rules 260

- TabNavigator tag, syntax 262

- tags

- allow-access-from 830
 - for metadata 369
 - See also* specific tag names

- Text control

- about 82
 - example 83
 - HTML text and 84
 - sizing 85
 - text 83

- Text tag, syntax 85

- text, dynamic ToolTip 546

- TextArea control

- about 85
 - example 86

- TextArea tag, syntax 86

- TextInput control

- about 88
 - binding and 89
 - example 89

- TextInput tag, syntax 89

- theme property 432

- themes 432

- about 432
 - Application tag attribute 432
 - compiling 800
 - definition 429

- limitations 433

- styles 410

- this keyword, event handlers 342

- tick marks

- definition 507
 - ranges 482

- tickPlacement property 507

- Tile container

- child sizing and positioning 238
 - default properties 237
 - example 238
 - horizontal 237

- Tile tag, syntax 239

- TileList control

- examples 123
 - keyboard navigation 125
 - user interaction 125

- title bar 196

- titleStyleDeclaration 416

- TitleWindow class, syntax 246

- TitleWindow container

- close button event 242
 - default properties 240
 - PopUp Manager and 243
 - popupWindow() method 240, 242, 561

- TitleWindow control, styles 416

- ToolTip Manager 543

- toolTip property 108, 109, 110, 382, 540

- ToolTips

- delay times 544
 - enabling and disabling 543
 - Hslider and 60
 - maxWidth property 542
 - Vslider and 60

- trace() function 748

- TraceOutputFileEnable 750

- TraceOutputFileName 750

- Tree control

- events 137
 - keyboard control 140
 - keyboard editing, events 139
 - keyboard editing, labels 139
 - nodes, editing and expanding 138
 - root nodes 136
 - TreeNode class and 103
 - XML data for 102, 136

- Tree node, syntax 141

- Tree tag

- drag and drop syntax 567
 - syntax 140

- TreeDataProvider interface, API 102
- Trial Edition 840
- troubleshooting
 - debugging settings 810
- type selectors
 - about 403
 - multiple 404

U

- UIComponent class, syntax 18
- UIObject class
 - parentApplication property 333
 - parentDocument property 332
 - syntax 18
- Unicode, shortcuts 428
- unicode-range attribute 427
- UnicodeTable.xml 428
- unnamed data services 658
- updating Flash Player 858
- upgrading Flex 840
- URL parameters 859
- URLs, callback 672
- user experience, instantiation order 577
- user_classes
 - directory 797
 - distributing components 841
- utility classes, Delegate 343

V

- validating data 629
- validation
 - complex objects 710
 - credit cards 713
 - currency 712
 - custom 706
 - data models 651
 - dates 715
 - disabling validator 711
 - e-mail addresses 717
 - form data 223, 224
 - forms 707
 - multiple fields 704
 - numbers 718
 - phone numbers 719
 - Social Security numbers 720
 - standard validators 712
 - strings 721
 - ZIP codes 722
- validators, List-based controls and 390

- VBox container. *See* Box container
- VBox tag, syntax 211
- VDividedBox container. *See* DividedBox container
- VDividedBox tag, syntax 215
- version detection
 - configuring 854
 - disabling 854
- version, Flash Player auto-update 857
- versionChecked 857
- verticalAxis 480
- verticalAxisStyle 501
- verticalFill 519
- verticalStroke 519
- VGA name color format 400
- ViewStack container
 - as data provider 100
 - child initialization order 251
 - default properties 248
 - example 249
 - resizing rules 248
 - sizing children 251, 261
- ViewStack tag, syntax 253
- VRule control
 - about 53
 - example 54
 - sizing 55
 - styles 56
- VRule tag, syntax 56
- VSlider control
 - about 56
 - events 58
 - example 57
 - keyboard navigation 60
 - labels 57
 - multiple thumbs 59
 - slider thumb 57
 - tick marks 57
 - tooltips 60
 - track 57
- VSlider tag, syntax 61

W

- warnings
 - about 750
 - showing override 811
- watchpoints 761
- web applications
 - about 840
 - deploying 836
 - See also* applications

- web services
 - access control 688
 - authentication 693
 - RPC-oriented, document-oriented 682
 - SOAP headers 683
 - stateful 683
 - whitelists 688
 - WSDL 682
- web-service-proxy-debug 747
- web.xml file
 - configuring 838
 - filter mappings 838
 - servlet mappings 839
- whitelists
 - data services 687, 701
 - definition 824, 831
- Window-less environments 798
- Windows, default browser 755
- windows-auto-install 856, 857
- WipeDown effect 464
- WipeLeft effect 464
- WipeRight effect 464
- WipeUp effect 464

X

- x-axis 481
- x-height, style property 398
- x-shockwave-flash MIME type 853

Y

- y-axis 481

Z

- ZIP codes, validating 722
- Zoom effect 464
- zooming components 458